Controller and estimator for dynamic networks

Amos Korman, Kutten Shay

Controller and Estimator for Dynamic Networks

Amos Korman * Shay Kutten†

Abstract

Afek, Awerbuch, Plotkin, and Saks identified an important fundamental problem inherent to distributed networks, which they called the *Resource Controller* problem. Consider, first, the problem in which one node (called the 'root') is required to estimate the number of events that occurred all over the network. This counting problem can be viewed as a useful variant of the heavily studied and used task of topology update (that deals with collecting *all* remote information). The *Resource Controller* problem generalizes the counting problem: such remote events are considered as requests, and the counting node, i.e., the 'root', also issues *permits* for the requests. That way, the number of requests granted can be controlled (bounded).

An efficient Resource Controller was constructed in the paper by Afek et al., and it can operate on a dynamic network assuming that the network is spanned by a tree that may only grow, and only by allowing leaves to join the tree. In contrast, the Resource Controller presented here can operate under a more general dynamic model, allowing the spanning tree of the network to undergo both insertions and deletions of both leaves and internal nodes. Despite the more dynamic network model we allow, the message complexity of our controller is never more than the message complexity of the more restricted controller.

All the applications for the controller of Afek et al. can be used also with our controller. Moreover, with the same message complexity, our controller can handle these applications under the more general dynamic model mentioned above. In particular, the new controller can be transformed into an efficient *size-estimation* protocol, i.e., a protocol allowing all nodes to maintain a constant factor estimation of the number of nodes in the dynamically changing network. Informally, the resulting new size-estimation protocol uses $O(\log^2 n)$ amortized message complexity per topological change (assuming that the number of changes in the network size is "not too small"), where n is the current number of nodes in the network.

Keywords: dynamic networks; controller; size-estimation; majority commitment; dynamic labeling schemes; name-assignment

*CNRS and Université Paris Diderot - Paris 7, France. E-mail: amos.korman@liafa.jussieu.fr. Supported in part by the ANR project ALADDIN, by the INRIA project GANG, and by COST Action 295 DYNAMO.

†Faculty of IE&M, Technion, Haifa 32000, Israel. E-mail: kutten@tx.technion.ac.il. Supported in part by a grant from the Israel Science Foundation.

1 Introduction

In common centralized settings, an online algorithm must make decisions based on past information, without knowing what the future holds. The main characteristic of a distributed network environment is an additional kind of uncertainty, namely, some nodes may need to make decisions without knowing what already happened in remote locations. Handling the common situation, where both kinds of uncertainties exist, is a challenging task which is yet far from being well-understood.

One should stress that the study of each of the above sources of uncertainty separately has been very extensive. In particular, the problem of *updating*- learning what happened in remote places- may be the main type of distributed algorithms actually used in networks. This is because after such information has been learned, the distributed problem is reduced to a better understood centralized one. For example, when a network node has learned the current topology of the network graph, it can compute the best routes to remote nodes by applying (the non-distributed) Dijkstra's shortest path algorithm [12] on the graph represented in the node's own memory. This approach is the one used in the Internet, see, e.g., the OSPF protocol [34].

This paper addresses problems affected by both kinds of uncertainties. Our main focus is on the (M, W)-*Controller* problem (introduced in [4]) which is considered as one of the elementary and fundamental tools in distributed computing (see e.g., [6]). Informally, the problem is defined as follows (a formal definition appears in Section 2.2). When an event "wishes to occur" at some node, the algorithm has to move either a *permit* or a *reject* to the node requesting the event. If the requesting node receives a permit, the permit is consumed at the node and subsequently, the event occurs. The key characteristics of an (M, W)-Controller are the constraints it imposes on the number of permits granted. Specifically, an (M, W)-Controller guarantees that the total number of permits granted is at most M. However, if a request is *rejected*, then at least $M - W$ permits are granted eventually.

Note, that the message complexity of a trivial controller can be very high. That is, if the only case a permit is moved is directly from the root to the requesting node, the message complexity can reach $\Omega(nM)$, i.e., $\Omega(n)$ per request. On the other hand, if all the requests are known in advance to one of the nodes, it is not hard to design an offline centralized algorithm whose message complexity is $O(n)$ (though the message size may be large).

The controller of [4] was designed to work under the *controlled* dynamic model, in which the topological changes do not occur spontaneously. Instead, when an entity wishes to cause a topological change at some node u, it enters a *request* at u, and performs the change only after the request is granted a permit from the controller. This model was rather visionary; today's Peer to Peer applications that did not exist then and, more generally, the now popular overlay networks, come to mind as examples of networks where topological changes can be controlled (we discuss this subject in Subsection 1.1). In [4], it was assumed that the network was spanned by a tree that may only grow, and only by allowing leaves to join the tree. I.e., only one type of topological changes is allowed: an insertion of a leaf node. Assuming that dynamic model, the message complexity of their controller is $O(N \cdot \log^2 N \cdot \log \frac{M}{W+1})$, where N is the total number of nodes ever to exist in the network.

In contrast, the resource controller presented here (still in the controlled model) can be applied under the more general dynamic model, allowing the spanning tree of the network to undergo both insertions and deletions of leaves as well as insertions and deletions of internal nodes. Despite the more dynamic network model we allow, the message complexity of our controller is never more than the message complexity of the more restricted controller.

Intuitively, it is not clear how to adapt the previous controller of [4] to handle these additional

topological changes efficiently. The reason is that the previous controller is based on storing permits at very specific depths of a spanning tree. Each node has a bin that may store permits. The bins are organized according to an underlying structure called *the bin hierarchy*. This bin hierarchy is employed in order to preserve some "sparseness" properties of the distribution of currently ungranted permits which are essential for the analysis. Specifically, in the bin hierarchy, each bin b at a node v has a *level* and a *size* which are determined by the precise distance from v to the root. Bin b also has a *supervisor* bin $sup(b)$ whose location with respect to b also depends on the precise distance from v to the root. A request always walks to a nearby bin b to obtain a permit. If that bin is empty, then the bin replenishes itself from its supervisor bin $sup(b)$ in the bin hierarchy. If $sup(b)$ is also empty then $sup(b)$ tries to replenish itself with permits taken from its own supervisor $sup(sup(b))$, etc. It follows, that the behavior of a node depends strongly on its precise distance to the root. The type of topological changes considered in [4] (the insertion of a leaf node) was allowed there since it did not affect the depths of the existing nodes and, therefore, did not affect the locations and sizes of the existing bins. However, in the more general dynamic model, a single change in the topology may change many of the above distances, thus destroying the beautiful combinatorial structure. For example, an insertion of an internal node may move many bins further away from the root without them even knowing that fact.

The controller presented here does not rely on a global predetermined structure. Instead, it strives to find and change the distribution of the currently ungranted permits as locally as possible.

1.1 Motivating the model

As in [4], we too assume the *controlled dynamic* model, in which a topological change does not happen instantaneously. Instead, it is delayed until getting a permit to do so from the resource controller. (See the model, Section 2.1.) Such a model may be found useful e.g. in overlay networks. Consider, for example, a Peer to Peer (P2P) network which is dedicated to users who are interested in a certain subject. When a user becomes uninterested in the subject, its node leaves the network. Hence, it is possible to request that it leaves in a "graceful" manner (i.e., that it first moves certain information to other nodes, in order to limit the disturbance its departure causes the network). Similarly, when a user becomes interested in the subject, its node joins the network in a "graceful" manner. In this context, it seems reasonable that the change can be delayed (beyond the inherent delay), so that our controller could be applied.

One can use, for example, a controller at a layer above the overlay network (and below the application of that network). This layer would present to the application a more orderly overlay network, one for which the number of nodes is known (and can be controlled), nodes are labeled economically to support an efficient routing scheme, some queries can be answered (e.g., a query about the lowest common ancestor of given two nodes in a tree), etc.

As mentioned, this paper does not handle spontaneous crashes. However, this seems to be less crucial in the case of overlay networks, where many topological changes are decided upon by designers or by algorithms (though, of course, sudden leaves are possible too).

From the theoretical point of view, the controlled dynamic model lies between two extremes. On one extreme, lies the static model (used in many studies) as well as the *one-by-one* model, where the topological changes are assumed to be sufficiently spaced so that the computations that follow one topological change are completed before the next topological change occurs (see, e.g. [20, 25, 29]). On the other extreme, lies the "chaotic" fully asynchronous and adversarial model. The middle ground controlled model has also a theoretical appeal, especially since many problems cannot be solved, or

can only be solved partially in the "chaotic" model. For example, inserting internal nodes in a rapid succession may prevent any message from reaching its destination in the chaotic model, thus making updates impossible.

1.2 Related problems

We now give a list of related problems, whose efficiency is evaluated by the message complexity measure. In Section 5 we show how to solve these problems efficiently in our dynamic model using our controller. For any given time, let n denote the current number of nodes in the network.

- The *size-estimation* problem [4, 20, 25]: Let $1 < \beta$ be some constant. In this variant of the updating problem, topological changes occur at different nodes of a network and it is required to maintain at all the nodes a β-approximation to the number of nodes in the dynamic network. I.e., it is required that each node v holds a variable $\tilde{n}(v)$ such that at all times, $n/\beta \leq \tilde{n}(v) \leq \beta n$.
- The *name-assignment* problem [4, 20]: in this problem, it is required to maintain a short unique identity $id(v)$ at each node v of the dynamic network (i.e., at any given time and for every two nodes u and v in the current network, we have $id(u) \neq id(v)$). By short, we mean that at any given time t, each identity is a number which is at most some constant times n. I.e., each identity is encoded using $\log n + O(1)$ bits.
- The *heavy-child decomposition* problem [22, 23, 25]: this problem is defined over dynamic trees. Here, it is required that each non-leaf node v holds some pointer $\mu(v)$ (that may change occasionally) to one of its children in the dynamic tree T. The pointed child is called *heavy*, and every other child is called *light*. The important feature of a heavy-child decomposition is that, at any given time t, each node has $O(\log n)$ ancestors (in T) which are light. The version of this problem on static trees was suggested by Tarjan [36] and used in multiple papers, both in the centralized setting and the distributed one (e.g., [8, 15]).

1.3 Other related work

The problem of counting the number of nodes in a growing tree network was suggested (but not solved efficiently) in [32] in the context of solving the *Majority Commitment* problem in a network where some of the nodes failed before the algorithm started [9, 14, 32]. Indeed, the state of the art solution for this problem employs a size estimation protocol for growing trees [7].

In general, the design of distributed protocols for problems in dynamic networks, and in particular, in dynamic trees, has been an important theoretical issue in distributed computing [2, 3, 5, 13, 35]. In particular, the problem of maintaining routing schemes and other informative labeling scheme representations dynamically was studied in [5, 20, 22, 23, 25, 26, 27, 28].

The controller problem bears similarities to the k-server problem introduced originally in the centralized setting and translated later to the distributed setting [11, 33]. There, *mobile servers* reside in some nodes. When a *request* arrives at some node v, the algorithm must decide which server δ should be moved to node v. Server δ cannot serve a later request that arrives at another node u without first moving from v to u. The *cost* for moving a server from v to u is the distance of v from u. The total cost of an execution (the *move* complexity) is the sum of the costs of the moves. The main difference between the k-server problem and the controller problem is that in the latter, multiple "servers" (permits) may be moved together without increasing the cost. In addition, here, the "server" is consumed

by the request. Finally, here, a request can also be rejected. We treat a reject rather similarly to a permit. That is, a move of a "reject" is also counted in the move complexity. (However, the number of rejects is not bounded.)

1.4 Our contributions

In this paper, we consider dynamic networks which are spanned by a spanning tree that may undergo both additions and deletions of both leaves and internal nodes. For such dynamic networks, we establish (M, W)-Controllers which are efficient in terms of their message complexity, where the message complexity is defined as the maximum total number of messages sent by the controller, taken over all possible executions.

Motivated by similarities to the k-server problem, we first consider the centralized setting and present two (M, W)-Controllers for it, which are efficient in terms of their move complexity. We would like to point out that although the centralized setting is simpler than the distributed one, the only known centralized controller with non-trivial move complexity is the centralized version of the distributed controller of [4], which supports only topology changes restricted to insertions of leaf nodes. The first centralized controller we present has move complexity $O(n_0 \log^2 n_0 \cdot \log \frac{M}{W+1} + \sum_j \log^2 n_j \cdot \log \frac{M}{W+1})$, where n_j is the number of nodes when the j'th topological change takes place, and n_0 is the initial number of nodes in the graph. The second centralized controller has move complexity $O(N \cdot \log^2 N \cdot \log \frac{M}{W+1})$, where N is the maximum number of nodes ever to exist simultaneously in the network.

We then consider the distributed setting and translate the first centralized controller to a distributed one[1]. The message complexity of the resulting distributed controller is asymptotically the same as the move complexity of our first centralized controller, namely $O(n_0 \log^2 n_0 \cdot \log \frac{M}{W+1} + \sum_j \log^2 n_j \cdot \log \frac{M}{W+1})$. Let us note that the asymptotic message complexity of the distributed controller presented here is never more than the message complexity of the more restricted controller in [4]. However, we assume that each message is encoded using $O(\log N)$ bits, while the controller in [4] assumes that each message contains only $O(\log \log N)$ bits.

Let us note that it was assumed[2] in [4] implicitly, that M is not "much larger" than polynomial in n_0, specifically, $M = {n_0}^{O(\log n_0 \log \log n_0)}$. This assumption hardly makes any difference as the typical applications of a controller consider M polynomial in n_0 anyway (for example, the controller used in the applications solving the size-estimation and name-assignment problems is employed with $M \leq n_0$). We use a similar assumption and require that $M = {n_0}^{O(\log^2 n_0)}$. If one wishes to give this assumption up then, similarly to [4], an extra additive term of $O(n_0 \frac{\log M}{\rho} \log \frac{M}{W+1})$ should be added to the message complexity, where ρ is the maximal number of bits used in a message ($\rho = O(\log \log N)$ in [4] and

[1] We would like to point out that the description of our centralized controller is done in a way that will ease the description of the distributed one. Indeed, the main combinatorial ideas are exposed already in the description of our centralized controller, and what is left in the distributed case is to deal with concurrency issues rising from parallel requests which arrive at roughly the same time as different nodes.

[2] To see this implicit assumption, consider a static scenario where all the requests are non-topological (in particular, the number of nodes remains n_0 throughout the execution). The (M, W)-controller presented in [4] operates in $O(\log \frac{M}{W+1})$ iterations, where in the i^{th} iteration, an $(M_i, M_i/2)$-controller is invoked on the tree (see Section 5 in [4]). At the beginning of the i^{th} iteration, the exact value of M_i should be known to all the nodes. This M_i value may vary from execution to execution, and cannot be known in advance. Moreover, M_i is encoded, typically, using $\Omega(\log M)$ bits. Therefore, in order to deliver the exact value of M_i to all n_0 nodes, $\Omega(n_0 \log M)$ bits should be sent per iteration. If every message is encoded using ρ bits, then $\Omega(n_0 \frac{\log M}{\rho})$ messages are required per iteration and an extra $\Omega(n_0 \log \frac{M}{W+1} \frac{\log M}{\rho})$ additive term should be added to the message complexity. The assumption follows from forcing this extra term to be asymptotically dominated by $O(n_0 \log \frac{M}{W+1} \log^2 n_0)$ and by taking $\rho = O(\log \log n_0)$, as claimed.

$\rho = O(\log N)$ in this paper).

Under the more general dynamic model, our controller can solve the size-estimation and name-assignment problems using message complexity $O(n_0 \log^2 n_0 + \sum_j \log^2 n_j)$. Moreover, one can transform our size-estimation protocol into a protocol that maintains a heavy-child decomposition of the spanning tree; the resulted protocol uses the same asymptotic message complexity, i.e., $O(n_0 \log^2 n_0 + \sum_j \log^2 n_j)$. All these applications use message size $O(\log n)$, where n is the number of nodes in the graph at the time the message is sent.

We also use the controller, especially as a size estimation protocol, for extending many existing labeling schemes supporting different queries (e.g routing, ancestry, etc.) so that these schemes can now operate correctly also under more general models. These extensions maintain the same asymptotic sizes of the corresponding labels (or routing tables) of the original schemes and incur a relatively low message complexity. (See Section 5 for more details.)

We recall that asynchronous size estimating was introduced by Bar Yehuda et al. [9] as a method for achieving a majority commitment (including asynchronous two phase commit) or an agreement, in networks many of whose nodes may have not woken up, or may have failed initially. By generalizing the size estimation protocol, the current paper also generalizes the majority commitment protocol to networks with additional scenarios (e.g., of nodes leaving, or joining as internal nodes).

The (M, W)-Controller presented in this paper was already used in the main algorithm in [22, 23, 28].

1.5 Paper outline

The paper is organized as follows. The model, the problem and the intuition behind the solutions are presented in Section 2. In Section 3, we describe and analyze the controllers in a centralized form. These descriptions also serve as a high level description of the distributed controller, and expose the main ideas behind it. Section 4 describes the distributed implementation, and analyzes the distributed controller by reducing it to a centralized one. The discussion regarding the applications of the new controller appears in Section 5.

2 Preliminaries

2.1 The model

2.1.1 Distributed and non-distributed algorithms

Except for the definition of topological changes (please see below in this section), we consider the standard point-to-point message passing asynchronous communication network model. The network topology is described by a general undirected graph (V, E), where the vertices represent processors and the edges represent bidirectional communication channels operating between neighboring nodes. The messages, which are transmitted over the links of the underlying network, incur an arbitrary but finite delay. We assume that each message is encoded using $O(\log N)$ bits, where N is a bound on the maximal number of nodes existing simultaneously in the network. Counting the number of messages is the main cost measure used in this paper. (We also use the measure of memory used per node).

As an intermediary step in constructing the distributed algorithm, we also construct a non-distributed algorithm. We refer to it as the "centralized algorithm." (In other words, this is a traditional non-

distributed non-parallel algorithm, sometimes also called "sequential"). We are not interested in the (computational) complexity of the centralized algorithm. However, we define for it a cost measure, called the "move complexity" (similar to the cost measure used for the centralized k-server problem), that evaluates the quality of the solution for the specific problem solved by the controller. The definition of the move complexity appears after the definition of the problem. Our main interest in the move complexity lies in the fact that it translates to the message complexity when we transform the centralized algorithm into a distributed one.

Finally, in transforming the centralized algorithm into a distributed one, we found it convenient to use the algorithm description method of [19] where messages are replaced by mobile agents. This model is equivalent to the standard point to point asynchronous network model. Additional details are given in Section 4.1 that deals with the distributed implementation.

2.1.2 Additional definitions

Given a rooted tree T, and a node $v \in T$, the depth of v is the hop distance between v and the root r of T. The ancestry relation is defined as the transitive closure of the parenthood relation, in particular, a node is its own ancestor.

We assume that throughout the dynamic scenario, the network is spanned by a rooted spanning tree T whose root r is never deleted. The tree T may undergo the following types of topological changes.

- **"Add-leaf":** A new degree one vertex u is *added* as a neighbor of an existing vertex v. The new node u is then considered a child of v in the spanning tree T.
- **"Remove-leaf":** A (non-root) vertex v of degree one is *deleted*.
- **"Add internal node" (between neighbors v and w in T):** Edge $e = (v, w)$ is split into two edges (v, u) and (u, w) for a new node u. If v was w's parent, then u is considered a child of v and w is considered a child of u.
- **"Remove internal node":** A (non-root) node u whose degree in T is larger than one is deleted together with all its non-tree edges. The nodes which were u's children in the tree, become the children of u's parent.

We note that we also allow events of the form "add a non-tree edge" and "remove a non-tree edge". Such events affect the topology of the network but not the topology of the spanning tree T. Since all the messages sent by our controller are sent over the edges of T, the communication network used by our controller (and also the one used by the controller of [4]) is not affected by such events. For this reason, we consider such events as non-topological. In addition, we also allow to insert an internal node u inside a non-tree edge (v, w), since this topological change can be considered as composed of adding u as a leaf of v and then adding the non-tree edge (u, w).

Recall that, as in [4], we assume the *controlled* model in which the topological changes do not occur spontaneously. Instead, when an entity wishes to cause an event (including a topological change) at some node u, it enters a *request* at u, and performs the event only after the request is granted a permit from the controller. A request to delete a node u arrives at u. A request to add a node arrives at the node's parent to be. A request to add a non-tree edge arrives at either one of its endpoints. When granted the permit, the requesting entity in the requesting node u is assumed to perform the topological

change after finite time. The particular way in which the topological change is made is beyond the scope of this paper, however, it is assumed that the topological change is made in a "graceful" manner. In the distributed setting, this means that (1) no messages are lost, and (2) any data belonging to the algorithm, stored at a node u that is about to be deleted, is moved to u's parent.

Incoming and outgoing links from every node are identified by so called *port-numbers*. It is maintained throughout the dynamic scenario, that at any given moment, the port numbers at each vertex v are distinct. We assume that at any time during the execution, each node knows the port number leading to its parent in the spanning tree T. When a new edge is attached to a node v, the corresponding port at v is assigned a unique (among v's ports) port-number. Different assumptions and methods by which a node can make a distinction between its own ports use different sizes of memory. To make our results applicable for a wide range of models, we assume the relatively wasteful model in which the port numbers are assigned by an adversary. We assume that the port numbers given by the adversary are encoded using $O(\log N)$ bits.

2.2 Controllers

The input of a controller arrives online in the form of *requests* arriving at arbitrary nodes. When a request arrives at a node u, the controller responds eventually by either granting it a permit or denying it, by delivering it a reject. In an attempt to do so, when a request arrives at a node u, the node triggers a distributed protocol that may exchange messages between neighboring nodes. An (M, W)-Controller is evaluated by its *message complexity*, i.e., the total number of messages it sends.

The (M, W)-Controller must not issue more permits than some given amount M. On the other hand, if some request is rejected then at least $M - W$ permits must be issued to requests eventually, even if these permits are issued physically to some nodes only after a reject is delivered to some other node. (The parameter W can be considered as the 'waste' in permits). To sum up, an (M, W)-Controller guarantees the following conditions.

Correctness conditions

- **Safety:** The total number of requests that were granted permits is at most M.
- **Liveness:** Every request receives either a permit or a reject eventually. If a request is rejected, then the total number of requests that will be granted a permit eventually is, at least, $M - W$.

In fact, the (M, W)-Controller presented here as well as the one in [4], solve the following very related (and at least as hard[3]) problem. Initially, an (M, W)-Controller in a directed spanning tree T has a set of M *permits*, and an infinite set of *rejects*, both considered to reside at the tree's root. Permits and rejects can be moved by the controller to other nodes. When a request arrives at a node u, the controller responds eventually by delivering either a permit or a reject to the request. The delivered object (a permit or a reject) is one of those that originally resided at the root and is currently residing in u. The delivery to the request consumes the object. A set (even infinite, in the case of rejects) of objects may be moved from a node to one of its neighbors in one message. An (M, W)-Controller must satisfy the above mentioned correctness conditions. (Observe that in this definition for the problem, the safety condition is guaranteed trivially.)

[3]Note that any solution to this related controller problem can directly be used to obtain a solution to the original controller problem with the same message complexity.

It is important to point out that this related problem also makes sense in the centralized environment, in a setting similar to that of the k-server problem. In that setting, an (M, W)-Controller is evaluated by the *move complexity*, i.e., the total number of moves of sets of objects, where each move is from a node to its neighbor. (This complexity resembles the move complexity in the k-server problem, on the one hand, and the message complexity of the distributed controller, on the other hand).

As mentioned before, we assume that $M = n_0^{O(\log^2 n_0)}$, where n_0 is the number of nodes at the beginning of the execution. If one wishes to avoid this assumption, then, as should be the case in [4], an extra additive term of $O(n_0 \cdot \log \frac{M}{W+1} \cdot \frac{\log M}{\rho})$ must be added to the message complexity of our distributed (M, W)-Controller, where ρ is the maximal message size. (As mentioned, this assumption hardly makes a difference since the typical applications of a controller consider an M that is linear in n_0, anyway.)

We recall that a controller may also control and count any type of non-topological event, (e.g., sales of tickets by different nodes, or even the number of messages sent by some other protocol [4]). The description in this paper emphasizes the measures the controller uses to handle topological changes. This is because this is the more challenging task. Indeed, an (M, W)-Controller that can handle the topological changes considered here can be easily transformed to an (M, W)-Controller that can handle these topological changes as well as non-topological ones. (The idea behind this transformation is the following: whenever a request to perform a non-topological change arrives at a node u, the node simply considers the request as a topological request for inserting a leaf node v.) Moreover, any algorithm to control other types of events (non-topological) in a dynamic network would have to deal with the topological changes and the dynamic nature of the network. For example, consider again the case that a protocol sends a message from some node u to some other node v. If an adversary can insert internal nodes in an uncontrolled way, the message may never arrive at its destination. Hence, even a controller for a non-topology related events must somehow deal with (and possibly control) the topological changes. The (M, W)-Controller described in this paper deals with all types of arriving requests together. That is, the Correctness conditions described above, control and bound the total number of granted requests, no matter which type of events they correspond to.

Terminating controllers: We now define another variant of the (M, W)-Controller called a *terminating (M, W)-Controller* which may be more useful in several cases. The definition of a terminating (M, W)-Controller is the same as that of an (M, W)-Controller except for the following. First, no rejects are given. Second, if at least M requests arrive, then the protocol must signal termination eventually (say, by outputting some termination signal at the root). If the protocol does not terminate eventually (in particular, this means that less than M requests arrive), then all the arriving requests must receive permits eventually. On the other hand, if the protocol terminates at time t, then from that time on, no permit is granted. Moreover, by time t, it is required that the number of requests that where given a permit is some m, where $M - W \leq m \leq M$. Finally, it is required that if the controller terminates, then by that time, all the requested events that received a permit, have occurred.

The (non-terminating) (M, W)-Controllers that we construct in this paper all use the same procedure regarding rejects. Specifically, in the distributed setting, if more than M requests arrive then the root broadcasts a "reject signal" to all nodes, eventually. In turn, a node that receives the "reject signal" rejects all subsequent incoming requests. On the other hand, a node does not reject requests until it receives a "reject signal". (In the centralized setting, our controllers use the same procedure except that the broadcast of the "reject signal" is simulated centrally and instantaneously). Such a controller (either distributed or centralized) is called an (M, W)-Controller with a *reject-wave*. The observation below shows that such a controller can be transformed easily to a terminating one.

Observation 2.1. *Any distributed (respectively, centralized) (M, W)-Controller with a reject-wave can be transformed into a distributed (resp., centralized) terminating (M, W)-Controller which has the same asymptotic message complexity (resp., move complexity) as the (M, W)-Controller.*

Proof: To prove the observation we consider the distributed setting; the centralized case follows analogously. Consider any distributed (M, W)-Controller with a reject-wave, called π. We show how to transform π to a terminating (M, W)-Controller, called π'. To obtain π', we first run π (including broadcasting the "reject signal"), except that nodes refrain from giving rejects to requests (even after receiving the "reject signal"). Instead, all the requests at any vertex v that should have been rejected in π are now put in a queue at v. In addition, in π', the broadcast of the "reject signal" will be followed by an upcast protocol as follows. Each vertex v that received the reject signal will first wait for a *termination* signal from all its children (clearly, a leaf doesn't need to wait for that). Consider all the events that correspond to requests at v that did receive permits. Node v now waits for these events to actually occur. Subsequently, if v is not the root then v sends a termination signal to its parent. Otherwise (v is the root), v simply terminates (say, by outputting some termination signal). It is easy to show that the resulting protocol π' is indeed a terminating (M, W)-Controller whose message complexity is higher than that of π by at most an additive factor which is linear in the number of nodes. ∎

2.3 Intuition and high level description

Initially, M permits reside in the root. Some of these permits may be grouped into packages of different sizes which can be moved from one place to another. In contrast to the controller of [4], our controller does not use predetermined locations, and the location of a package has nothing to do with its size. (In fact, a node may store several packages of different sizes).

If a request arrives at a node u having a 'small' package, then a permit from that package is granted to the request. Otherwise, u may contain a package that is "too large", or may not contain a package at all. In either of these cases, an agent is sent up the tree looking for the first package of some size x, located at a distance about x from u. (We use the terms "about" and "roughly" throughout this informal part, in order to expose the intuition better; the exact details of the algorithms appear in the next sections). If no such package exists on the way from u to the root then a package of the appropriate size (roughly the distance between u and the root) is created at the root, if enough permits to be included in the package still reside in the root. If there are not enough permits at the root, a reject signal is broadcast.

When an appropriate package P of size x is found (or created) at a node w (whose distance from u is also roughly x), its content is distributed along the path I from w to u, as follows. This x is roughly 2^i for some i (we call P a level i package). First, package P is moved to a node $v \in I$ at distance roughly 2^{i-1} above u, and then splits into two level $i-1$ packages. One of these packages remains at v and the other, P_2, is moved to some node $z \in I$ at distance roughly 2^{i-2} above u. P_2 is then split into two packages of level $i-2$, and the process continues until one level zero package is moved to u. One permit from this level zero package at u is then granted to the request. The other packages created in the described process wait for additional requests.

To bound the message complexity, we bound the moves of packages. Note, that a permit can be transferred from a package P to a package P' only if the size of P' is about one half of the size of P (a result of P's split). Hence, throughout the dynamic scenario, a permit may have belonged only to a logarithmic number of packages. Since a move of x permits is to a distance that is about x (about one

per permit in the package), the message complexity is low.

The safety condition is satisfied trivially, since the root does not issue more than M permits. In order to show that the liveness condition holds, we show that when a reject is issued, the total number of permits that remain in packages (and in the root) and are not granted to requests, is small. For that purpose, we associate each package P with a set of nodes (some of them may have been deleted already) called the "domain" of P. The domains satisfy the following invariants. (1) The size of P's domain is about the size of P, and (2) the domains of two packages of the same level are disjoint.

These invariants guarantee that at any time, the number of of permits "stuck" in packages of a given level (and size) is small. Summing over the levels yields an upper bound on the number of permits "stuck" in packages at that time. This yields a lower bound on the number of permits that are granted to requests eventually.

Let us hint how we ensure that domains have the above mentioned properties. Recall, that when a request arrives at a node u, an agent finds the smallest i such that there is a package P of size about 2^i at roughly distance 2^i above u. This means that for $j < i$, there are no level j packages at distance about 2^j above u. Hence, there is a path of size roughly 2^j at distance roughly 2^j from u, which is free from a level j package P'. Some nodes in this path may still belong to a domain of some level j package P', even though P' itself does not reside in that path. However, using a counting argument, we manage to locate a subpath of that path that is also of size roughly 2^j, and does not intersect any domain of any existing package of level j. This subpath defines the initial domain of the level j package, which is put at the top of that subpath, as a result of the recursive splitting of P.

When topological changes occur, we update the domains so that the above mentioned domain invariants are maintained. Updating of domains is done only for the sake of the analysis, therefore, in particular, the algorithm does not need to use any communication for updating domains and does not need to notify the nodes about their domain memberships.

3 The Centralized Controller

In this section, we consider the centralized setting and present our new (M, W)-Controller which is efficient in terms of its move complexity. Initially, a set of M permits and an infinite set of rejects reside in the root's *storage*. Requests arrive in an online fashion at different nodes, and the controller needs to move to each request either a permit or reject, such that the correctness conditions hold. In the next section, we show how to implement this controller distributively. The move complexity used here will translate later into the message complexity in the distributed setting.

3.1 The algorithm

As in [4], we too assume first that there exists a fixed and known upper bound U on the number of nodes ever to exist in the graph including the deleted nodes (in other words, U is a bound on the initial number of nodes n_0 plus the total number of additions of nodes). The removal of this assumption is done in Section 3.3, using a rather standard method which is similar to the one used in [4].

The algorithm uses a dynamic data structure called *packages*. Each package resides in some node which is referred to as the *host* node of the package. A node may store multiple packages. There are two kinds of packages, namely *permit packages* and *reject packages*. Each permit package contains some finite number of permits, and each reject package is interpreted as representing infinitely many of rejects.

(Rejects are identical to each other, so a reject package can be represented by a constant number of bits.) A permit package may be either *static* or *mobile*. Informally, a static (permit) package is used to grant requests for the node hosting it and a mobile (permit) package is used to deliver sets of permits from place to place. Each permit package (either static or mobile) has a *size*, which is the number of permits in the package. The size of a static package is between 1 and ϕ, where $\phi = \max\{\lfloor \frac{W}{2U} \rfloor, 1\}$, and the size of a mobile package is precisely $2^i\phi$ for some integer $i \geq 0$. Consider a mobile package of size $2^i\phi$. For convenience, we call i the *level* of the package. It will follow from the description of the algorithm, that the level of a (mobile) package is between zero and $\log U + 1$.

Initially, there are M permits and infinitely many rejects residing in the *storage* of the root, and no packages anywhere. The following actions are supported by the data structure:
(1) The creation of either a mobile or a reject package P residing in the root. The content of the package is taken from the root's storage.
- If a mobile package is created then it is created together with a size (which determines a level).
- If a reject package is created, it represents infinitely many rejects.
(2) The *split* of a package into two packages.
- When a mobile package of level $i > 1$ splits, both resulting packages are mobile packages of level $i-1$. When a mobile package of level 1 splits, one of the resulting packages is a mobile level zero package and the other is a static package containing ϕ permits.
- When a reject package splits, it splits into two reject packages (each representing infinitely many rejects).
(3) The *move* of a package from its host node to a new host node.
(4) The *granting* (respectively, delivering) of a permit (resp. reject) from a static (resp. reject) package in a node u to a request in the same node. The granting of a permit decreases the size of the static package by one. If, consequently, the size of the static package becomes zero then the package is canceled, i.e., it no longer exists in the data structure.

We need the following definitions. Let $\psi = 4\lceil \log(U) + 2 \rceil \cdot \max\{\lceil \frac{U}{W} \rceil, 1\}$. Given a node u and a given time t, a *filler* node w with respect to u is an ancestor w satisfying the following two conditions at time t:
a) w contains a mobile package P of level j.
b) if $j = 0$ then $0 \leq d(u, w) \leq 2\psi$, otherwise, if $j \neq 0$, then $2^j\psi < d(u, w) \leq 2^{j+1}\psi$.

We are now ready to describe Protocol GrantOrReject(u) which is applied by the algorithm in response to an arrival of a request at some node u.

Protocol GrantOrReject(u)

1. If there exists a reject package at u then the request is rejected. Otherwise the following happens.
2. If there exists a static package P residing in node u, then a single permit in P is granted to the request. Subsequently, the size of P is reduced by one. If, consequently, the size of P becomes zero then P is canceled. Consider the following cases.
   - If the request is for a topological event which is of type "remove leaf" or "remove internal node" and u, the vertex to be removed, holds one or more packages, then these packages are moved to u's parent. Subsequently, u is removed.
   - Otherwise, the requested event takes place when the request is granted the permit.

If there is no static package at u then the following happens.

3. (a) If there exists an ancestor of u that is a filler node with respect to u, then let $\rho(u)$ be such a filler node that is the closest to u (in the tree T). Also, let $P(u)$ and $j(u)$ be such that $\rho(u)$ has the package $P(u)$ of level $j(u)$ satisfying the conditions mentioned above, in the definition of a filler node. The content of Package $P(u)$ will now be distributed along the path to u as described in Item 4 below.

   (b) Otherwise (no filler node exists), let $j(u)$ be the smallest integer such that $d(u,r) \leq 2^{j(u)+1}\psi$. If there are less than $2^{j(u)}\phi$ permits left at the storage of the root then the request is rejected. In addition, a reject package is placed in every node in the network. This is done by first creating a reject package at the root, and then using splitting and moving. (I.e., to deliver a reject package from a node u to its d children, we first create d new reject packages at u using splitting and then deliver one of these new packages to each child.) If the request was not rejected, then the handling of the request proceeds as follows.

   A mobile package $P(u)$ of level $j(u)$ is created at the root r. Note, that at this point, the root becomes a filler node $\rho(u)$ with respect to u. The content of package $P(u)$ will now be distributed along the path to u as follows.

4. For each $k \in \{0, 1, 2, \cdots, j(u)-1\}$, let u_k be the ancestor of u satisfying $d(u, u_k) = 3 \cdot 2^{k-1}\psi$. Apply the following procedure PROC recursively such that initially, $P = P(u)$.

   Procedure PROC(P)**:** Given a package P of level k at a vertex w, act as follows.

   - If $k > 0$ then move P from w to vertex u_k. Then, split P into two packages P_1 and P_2, each of level $k-1$. Leave P_1 at u_k and apply PROC(P_2) recursively.
   - If $k = 0$ (including the case where $j(u) = 0$) then package P is moved to u and becomes static. The request is then granted to u from P according to item 2 above.

3.2 Correctness and Complexity

Given a time t, an *existing* package (respectively, node) is a package (resp., node) that exists in the graph at time t. Every existing mobile package P is associated with a set of (not necessarily existing) nodes, called the *domain* of package P, which may change from time to time. The domains are used for analysis purposes only, therefore, in particular, when a node joins or leaves some domain, no actions are required by the algorithm to support the change. The following invariants are maintained:

The domain invariants

1. For every k, the domain of each existing mobile package of level k contains $2^{k-1}\psi$ nodes.
2. For every k, the domains of the existing mobile packages of level k are disjoint.
3. At any time, the currently existing nodes in the domain of each mobile package form a path hanging down (away from the root) from some child of the node holding the package.

Below, we define the domains and show that the domain invariants hold. Initially, there are no packages and no domains. When a package is canceled or becomes static, its domain is canceled. Similarly, when

a package splits, its domain is canceled and new domains are given to the new packages that result from the split. A package is *formed* at some time t if, at time t, the package is either created (at the root) or results from a split. (Note that this happens only after a request arrives at some node u.) We first define the domain of a mobile package at the time it is formed.

Let $\rho(u)$ be the vertex defined in Item 3 of the description of the protocol above. Consider the following cases.

Case 1) If $j(u) = 0$ then no mobile package is formed, so no new domain is necessary. (This is true also for level zero packages in item 3.b, since there, even though a level zero mobile package is created at the root, it is then moved to u immediately and becomes static.)

Case 2) If $j(u) > 0$ then let $P(u)$ be the level $j(u)$ package residing in $\rho(u)$ (as defined in item 3). After the recursive procedure PROC($P(u)$) is completed, we have the following. For $k \in \{0, 1, 2, \cdots, j(u)-1\}$, one level k mobile package P_k is located at the node u_k above u, satisfying $d(u, u_k) = 3 \cdot 2^{k-1}\psi$. (Note, that one static package is located at u but for a static package we do not need to define a domain.) For every $k \in \{0, 1, 2, \cdots, j(u) - 1\}$, the domain $Dom(P_k)$ associated with the package P_k is the set of vertices x on the path connecting u and u_k which satisfy $1 \leq d(x, u_k) \leq 2^{k-1}\psi$. (Note that in the case where item 3.b is applied, we do not need to define a domain for $P(u)$ since it is split immediately after being created.)

Let us now define how the domain of an existing mobile package P may be affected by a topological event τ occurring in the (existing) nodes in $Dom(P)$.

Case 3) An event of type "add leaf" has no affect on $Dom(P)$.

Case 4) If τ is of type "add internal node", and u, the added vertex, is inserted as a parent of a node in $Dom(P)$, then u is added to domain $Dom(P)$ and the bottom most (farthest from the root) existing node in $Dom(P)$ is removed from P's domain.

Case 5) If τ is of type "remove leaf" or "remove internal node", and u, the removed node, belongs to $Dom(P)$, then u is deleted from the graph but still continues to belong to $Dom(P)$.

Claim 3.1. *The domain invariants hold at all times.*

Proof: Clearly, the invariants hold initially when there are no packages. Assume by induction, that they hold just before the next time t where either a package is formed or a topological event occurs.

First, consider the case that at time t, a package is formed. New domains are defined only due to an application of Procedure PROC($P(u)$). After Procedure PROC($P(u)$) is completed, for every $k \in \{0, 1, 2, \cdots, j(u) - 1\}$, one new level k mobile package P_k is located at vertex u_k above u, such that $d(u, u_k) = 3 \cdot 2^{k-1}\psi$. The first and third domain invariants follow directly from the description in Case 2 in the definition of the domains above. It is left to show that the second domain invariant holds.

Fix some $k \in \{0, 1, 2, \cdots, j(u)-1\}$. Let I_k denote that path containing the ancestors w of u satisfying $2^k\psi < d(u, w) \leq 2^{k+1}\psi$. The definition of $j(u)$ implies that just before PROC($P(u)$) is applied, there is no mobile package of level k in path I_k, including in particular, at any vertex in $Dom(P_k) \subset I_k$.

Therefore, by the third domain invariant, $Dom(P_k)$ does not intersect with any other domain of a package of level k residing in a *descendant* of u_k. By the first and third domain invariants, and by the fact that $I_k \setminus Dom(P_k)$ does not contain any mobile package of level k either, we obtain that $Dom(P_k)$ does not intersect any domain of any other level k package residing in an *ancestor* of u_k. Therefore, the second domain invariant also holds at time t.

Now, consider the second case where domains may change at time t, that is, a topological event τ occurs. No change in a domain is needed in Case 3 or when the topological event concerns nodes which are not in any domain and do not hold any package. Hence, assume that τ is of type "add internal node", and vertex u is added at time t between two existing vertices v and w. Case 4 above is applied. P's domain loses its bottom most node and gains node u that is new, and hence has not belonged to any domain in the level of P. Therefore, the first and second domain invariants continue to hold. In addition, the removal of the node from the domain does not disconnect the path which is $Dom(P)$ since the removed node is the bottom most. Similarly, the addition of the new node u keeps $Dom(P)$ as a path since u (and edges (v,u) and (u,w)) replace edge (v,w) on the path. Hence, the third invariant continues to hold.

Consider now the case where τ is of type "remove leaf" or "remove internal node". If just before time t, the removed node u contained several packages, then these packages are moved at time t to u's parent. Note that by the induction hypothesis on the third domain invariant, u does not belong at time t to the domain of any of these packages, therefore, its removal does no affect the domains of these packages. Therefore, the first and second domain invariant, with respect to these packages, continue to hold. Since u's children become the children of its parent, the third invariant holds as well. Consider, now, the case where just before time t, the removed node u belonged to $Dom(P)$ for some package P. In this case, u continues to belong to $Dom(P)$. Therefore, the first domain invariant holds. In addition, no node was added to any domain, therefore, the second domain invariant still holds as well. Since u's children become the children of its parent, the third invariant also holds. This completes the proof. ∎

3.2.1 Correctness

The fixed bin hierarchy of [4] trivially guarantees some invariants which are essential for the analysis therein and which are rather similar to the domain invariants. Thus, once we established the fact that the domains invariants hold at all times we can continue with the following lemma whose proof is rather similar to Lemmas 4.7 and 4.8 in [4].

Lemma 3.2. *The correctness conditions hold for the centralized controller.*

Proof: As mentioned before, since there are M permits initially residing in the storage of the root, the safety condition is trivially satisfied.

Now, consider the first time a request is rejected. Let us first bound the sum of the sizes of the currently existing mobile packages. By the first two domain invariants, the number of level k mobile packages is at most $\frac{U}{2^{k-1}\psi}$. Observe that $\frac{\max\{W/2U,1\}}{\max\{U/W,1\}} \leq \frac{W}{U}$. Therefore, the sum of the sizes of the level k mobile packages is at most,

$$\frac{2^k U}{2^{k-1}} \cdot \frac{\phi}{\psi} \leq \frac{2^k U}{2^{k-1}} \cdot \frac{W}{4U\lceil \log(U) + 2 \rceil} = \frac{W}{2\lceil \log(U) + 2 \rceil}.$$

By the first domain invariant, the domain of a level k package contains $2^{k-1}\psi$ nodes. Since $\psi \geq 1$, it follows and that $2^{k-1} \leq U$, and thus, $k \leq \log U + 1$. Hence, the number of levels is at most $\log U + 2$.

It follows that the sum of the sizes of all the mobile packages is at most $W/2$.

Let us now bound the sum of the sizes of the existing static packages. If $W < 2U$ then $\phi = 1$, hence, there are no static packages (once a static package of size 1 is formed, the single permit in it is granted immediately and the package is canceled immediately). If, on the other hand, $W \geq 2U$, then $\phi \leq W/2U$. Therefore, the sum of the sizes of the static packages is, at most, $U\frac{W}{2U} = W/2$.

It follows that the sum of the sizes of all the packages (both mobile and static) is at most W. Therefore, at any given time, the total number of permits in all the currently existing packages is at most W.

Consider now the first time a request is rejected at some node u. Then, according to Item 3b of Protocol GrantOrReject(u), the number of permits left at the root is less than $2^{j(u)}\phi$, where $j(u)$ is the smallest integer such that $d(u,r) \leq 2^{j(u)+1}\psi$. For analysis purposes, by using additional $2^{j(u)}\phi$ *imaginary* permits, we create the package $P(u)$ and distribute its content along the path from the root to u as described in Item 4, except that u is not granted a permit. At this (imaginary) point in time, using the same analysis as before, we get that the sum of the sizes of all the existing packages (both mobile and static) is at most W. Since at least M permits (imaginary and real) were issued by the root, it follows that at least $M - W$ permits were granted to requests. Since all the granted permits are real (non-imaginary), the liveness condition holds as well. This proves the lemma. ∎

3.2.2 Complexity

Lemma 3.3. *The move complexity of the centralized (M,W)-Controller is $O(U\frac{M}{W}\log^2 U)$.*

Proof: Permits and rejects move only in packages. When a reject is issued for the first time, a reject package is created at the root and a reject package is delivered to each node in the existing graph by means of splitting and moving. In particular, throughout the execution, each node receives at most one reject package. Consequently, the move complexity for all the reject packages is at most U.

A package may be moved up the tree only as a result of a deletion. Specifically, if a node u holds several packages and is given a permit to delete itself, then it first moves its packages to its parent (see item 2 in the algorithm). Since the number of deletions is at most U, the total move complexity resulting from such moves is U.

The only other moves of permit packages are as a result of applying procedure Proc(P). Therefore, throughout the scenario, each mobile package P moves at most twice. Once, when P is created during the application of some procedure Proc(P') and once, if P is the level k package $P(u)$ found at the filler node $\rho(u)$ and moves to u_k (if $k > 0$) or to u (if $k = 0$). Both of these cases of moves are to distance $O(2^k\psi)$ where k is the level of P. Note, that for any level k, each permit may have belonged to at most one level k mobile package. (The reason for that is that a permit may be moved from a package P' to a mobile package P'' only if P'' is one of the packages resulting from splitting P', in which case, the level of P'' is less than the level of P'.) Therefore, since at most M permits are issued[4], the total number of packages of level k ever to exist is at most $\frac{M}{2^k\phi}$. Consequently, the sum of the costs of the moves made by packages of level k is $O(2^k\psi\frac{M}{2^k\phi}) = O(M\frac{\psi}{\phi}) = O(U\frac{M}{W}\log U)$. Since there are $O(\log U)$ levels k, the lemma follows. ∎

The move complexity can be reduced further. The trick for that is the same as the one introduced

[4]Note that the packages described in the proof of the previous lemma which contain imaginary permits, do not exist in the real scenario, and only exist for analysis purposes; therefore they are not counted as a part of the move complexity.

in Section 6 of [4]. We explain it here for completeness. In addition, we hope that this can help the reader with the next subsection too, which uses a similar trick for a different purpose.

In order to deal with the cases where M/W is large, one can iterate the controller $O(\log \frac{M}{W+1})$ times. In each iteration, the "waste" is at least halved. First, set $M_0 = M$. In the i'th iteration, the controller is initiated with the parameters $(M_i, M_i/2)$. When the i'th iteration terminates, the algorithm counts the number L of unused permits in the packages that exist at that time. Then, instead of rejecting a request, the algorithm clears the data structure, sets $M_{i+1} \leftarrow L$ and starts the $i+1$'st iteration.

Consider, first, the case where $W > 0$. In this case, after $i' = O(\log \frac{M}{W+1})$ iterations, $M_{i'+1}$, the number of unused resources in the existing packages, is within a constant multiplicative factor of W and the $i'+1$ iteration (which is the last iteration) is initiated with parameters $(M_{i'+1}, W)$.

If $W = 0$, then the controller needs to grant precisely M permits. We first run the $(M, 1)$-Controller until it wishes to a assign a reject. Then, if the number of permits granted is M we are done. Otherwise, if the number of permits granted is $M-1$, we run the trivial $(1, 0)$-Controller (which has linear message complexity). We sum up the above discussion by the following observation.

Observation 3.4. *The move complexity of the centralized (M, W)-Controller is $O(U \log^2 U \log \frac{M}{W+1})$.*

3.3 The case that no fixed U is known

It was shown in Section 5 of [4] that a controller for the case where a fixed upper bound on U is not known can be obtained from one which is designed for the case where such a bound is known in advance. The proof of the following theorem relies on the same principles. Let us note, that the move complexity mentioned in the second part of the theorem is never (asymptotically) more than the move complexity mentioned in the first part of the theorem. We prove the first part of the theorem also since this part is translated to the distributed setting.

Theorem 3.5.
- *There exists a centralized (M, W)-Controller whose move complexity is $O(n_0 \log^2 n_0 \cdot \log \frac{M}{W+1} + \sum_j \log^2 n_j \cdot \log \frac{M}{W+1})$, where n_j is the number of nodes at the time the j'th topological change takes place.*
- *There exists a centralized (M, W)-Controller whose move complexity is $O(N \cdot \log^2 N \cdot \log \frac{M}{W+1})$, where N is the maximal number of nodes existing simultaneously in the network. (It is not required that N is known in advance).*

Proof: In the proof of the theorem, a controller for the general case is constructed by running the algorithm claimed in Observation 3.4 in iterations, using a different estimate U_i in each iteration.

Let us now construct a centralized (M, W)-Controller with the complexity stated in the first part. In the i'th iteration, we assume that $U = U_i$, and consider our (M_i, W)-Controller, where U_i and M_i are defined below. This (M_i, W)-Controller can be easily transformed into a terminating (M_i, W)-Controller as describe in Observation 2.1. In the i'th iteration, we execute the resulting terminating (M_i, W)-Controller.

The first iteration is initiated with $M_1 = M$, and with $U_1 = 2n_0$, where n_0 is the initial number of nodes in the graph. Let N_i denote the number of nodes in the graph at the beginning of the i'th iteration and let Y_i denote the number of (granted) requests during the i'th iteration. Let $U_i = 2N_i$. Let Z_i be the number of *topological changes* that occur during the i'th iteration. The i'th iteration is terminated when Z_i reaches $U_i/4$. Note, that N_i, Z_i and Y_i can be computed easily in the centralized

setting (later, in the distributed setting, we shall use a second controller to estimate Z_i, and shall count the exact values of Y_i and N_{i+1} at the end of each iteration). When the i'th iteration terminates, the data structure is first initialized, by removing all the existing packages from the graph. The $i+1$'st iteration is then initiated with $U_{i+1} = 2N_{i+1}$ and $M_{i+1} = M_i - Y_i$. Clearly, U_i satisfies $U_i/4 \leq n \leq U_i$ during the i'th iteration.

The fact that the modified algorithm implements an (M, W)-Controller is clear. For each iteration i, let j_i denote the number of topological changes that occurred from the beginning of the execution until the i'th iteration was terminated, and let $j_0 = 0$. Let s be the number of iterations. If $s > 1$ then for each $1 \leq i < s$, we have $N_i = \Theta(j_i - j_{i-1})$. For each $1 \leq i < s$, $U_i = \Theta(N_i) = \Theta(N_{i-1})$. By Lemma 3.4, the move complexity during the i'th iteration is $O(U_i \log \frac{M_i}{W+1} \log^2 U_i)$. Therefore, for $1 \leq i < s$, the move complexity during the i'th iteration is

$$O((j_{i-1} - j_{i-2}) \cdot (\log^2 U_i \cdot \log \frac{M_i}{W+1})) = O(\sum_{j=j_{i-2}+1}^{j_{i-1}} \log^2 n_j \cdot \log \frac{M}{W+1}) \ .$$

The move complexity during the first iteration is $O(n_0 \log \frac{M_i}{W+1} \log^2 n_0)$.

Hence, the total move complexity is $O(n_0 \log^2 n_0 \cdot \log \frac{M}{W+1} + \sum_{i>1} \sum_{j=j_{i-2}+1}^{j_{i-1}} \log^2 n_j \cdot \log \frac{M}{W+1}) = O(n_0 \log^2 n_0 \cdot \log \frac{M}{W+1} + \sum_j \log^2 n_j \cdot \log \frac{M}{W+1})$, as required in the first part of the theorem.

To construct the centralized (M, W)-Controller with the complexity stated in the second part of the theorem, we change the definition of an iteration as follows. An iteration is terminated (and a new iteration starts) only when the number of nodes is doubled from the maximal number of nodes existing simultaneously in the graph before the iteration started. The analysis is very similar to the one above. ∎

4 The Distributed Controller

As in the centralized setting, the main difficulty is the construction of an (M, W)-Controller, assuming a known upper bound U on the number of nodes ever to exist in the network (including deleted nodes). We first describe the scheme assuming the above assumption, and then, in Subsection 4.5, we sketch the (rather simple) removal of this assumption.

4.1 Overview

We note that the centralized controller was constructed in such a way that it can be implemented distributively. Moreover, the proof of the distributed version is by a reduction to the centralized one. Let us give here an overview.

The arrival of a request at a node u creates a mobile agent (see [19, 10]; in [19] this is called a "token") at u. If there is no static package at u then the agent climbs the tree (carried by messages) until it reaches a filler node or the root. It then takes the package P located there and performs PROC(P) by walking down the tree towards u.

The only real difference from the centralized setting is the fact that an agent cannot act instantaneously on multiple nodes, while in the centralized algorithm, each request is handled fully before the next request arrives. To ease the proof, the instantaneous action is simulated using locks, and using

the assumption that topological changes occur in a "graceful" manner (see Section 2.1, this issue is also discussed in detail in Section 4.2). That is, the agent starts from u and climbs (locking every node on its way) until it finds a filler node ρ with respect to u. On its way up, if the agent reaches a locked node, it first waits for the node to become unlocked and only then continues its action. If multiple agents wait at a locked node then, when the node becomes unlocked, the agent resuming its action, is the first one of them to arrive at the node. When the agent finds the filler node ρ, it acts on the packages data structure in the nodes on the path from ρ to u, simulating PROC. Since at that point, all the nodes on that path are locked, the agent's actions can be shown to be similar to those of the centralized algorithm. When the agent answers the request at u, it climbs up the tree again, until it reaches ρ, and then returns back to u. On its second trip down to u, the agent unlocks every node it passes.

We prove the distributed implementation by mapping each distributed execution to an execution of the centralized controller. In the simulation, a request that arrives at some time t_1 but is granted later at time $t_2 > t_1$ in the distributed execution, is mapped to a request that arrives and is granted at time t_2 in the centralized execution. (This is because the data structure in the distributed case can still change after time t_1, but at t_2, all its relevant parts are locked and cannot change.) We believe that given the above, the reader can construct the distributed implementation. For completeness, a detailed description is given below. The proof may be of interest in itself, as an approach for proving distributed algorithms by reducing them to centralized ones.

4.2 Issues regarding removals of nodes

Before dwelling into the details of the distributed implementation of the centralized controller, let us first discuss some issues regarding the removal of nodes in the distributed setting. As in the centralized setting, when a request to delete a node u is granted a permit, the requesting entity in the requesting node can then perform the topological change. However, in the distributed setting, we assume that the deletion is made in a "graceful" manner. In particular, (1) in the case of a node's deletion, any data belonging to the algorithm, stored at that node, is moved to the node's parent; (2) no messages are lost. (A message sent to a parent who is being deleted is either actually both sent to and received at the parent before the parent is deleted, or is sent to and received by the new parent).

Note, that ensuring the above "graceful" manner may require the use of some additional protocols, which may not be the same in different types of networks. For example, in the case of deleting some node u, the requesting entity may need to perform a handshake with the neighboring nodes to verify that no messages are on their way to u. Alternatively, a loss can be allowed temporarily, and some form of acknowledgment and retransmission may be used to recover from that loss. As another example, any addition of an edge may require a handshake between the endpoints of the edge (this, for example, is the case when the edge represents a TCP connection in an overlay network). Such additional protocols may be interesting in themselves. However, they are beyond the scope of the current paper.

Another issue regarding the deletion of a node is whether there are additional unfulfilled requests that arrived at that node from the environment, and are now waiting at the node for either a permit or a reject. As discussed later in Subsection 4.4.2, we do not consider the memory space these requests may occupy as a part of the space of the algorithm. Still, it may be possible for the algorithm to handle some requests of this kind. This is not necessarily possible (or even desirable) in all the cases. That is, some such requests may lose their meaning if the node is deleted, and the environment may no longer be interested in having them granted permits. (Examples are additional requests to delete u after it is already deleted.) There may be cases that some such requests are still meaningful and can be treated

even if the node is deleted. In such cases, the "graceful" manner assumption includes taking care of such requests. Then, the "graceful" manner includes identifying such requests and moving the requests to the parent node. Again, the way the deletion requesting entity handles this is beyond the scope of this paper. We note that, while the requirement for the "graceful" manner may delay the requesting entity in the actual deletion, we no longer count the node in the number of existing ones.

4.3 The Distributed Implementation

We now begin the description of the distributed controller. As mentioned, we first assume that a fixed upper bound U on the number of vertices ever to exist in the network (including the deleted vertices) is given in advance. The removal of this assumption is discussed in Appendix A.

For simplicity of presentation, we describe the distributed implementation using more than one protocol layers. To simplify the higher layer, the algorithm is written as an algorithm for a *mobile agent*. By doing that, we manage to push most of the less interesting details into the lower layer that supplies services to the agent, rather than to the actions of the agent. (This follows the algorithm description method of [19].)

4.3.1 The higher layer description

A mobile agent is a process that can move from a node to another node by issuing a move instruction. This instruction moves the agent process together with its state, including its program counter and its variables. Hence, the agent can read and write its own variables even after the move. While at a node, the agent can access information stored at the node. (This storage is called a *whiteboard* e.g. in [10]). The whiteboard is not moved when the agent moves. It remains at the node and can be read and written by any agent that visits it, but only while visiting the node. (Of course, an agent can copy the content of the whiteboard to a variable it carries with itself, but this may increase the communication complexity, since additional information is then carried.) To reduce issues related to concurrency within one node, only one agent is active at a node at one time. That is, the agent handles an event (e.g. the arrival at the node) atomically. After that, the agent either leaves the node, or waits, while other agents may arrive and handle their events. For additional definitions and implementation details see [19].

The arrival of a request at a node u (as an input from the environment) creates an agent at u. (As mentioned later, using "locks", we may assume that the next request at u may arrive only after the agent of the current request terminates.) This agent starts traveling in the tree and after some time returns to u with the answer for the request (either a grant, or a reject). We assume that when the agent returns to u, the agent knows how to communicate with the entity in u's environment that issued the request. This communication is needed in order to deliver the answer. This communication is beyond the scope of the current paper. (For example, the requests may be injected to u by the environment to a queue, and the request handled by the agent is the one at the top of the queue.)

As in the centralized algorithm, there are two kinds of packages, namely *permit packages* and *reject packages*. Each permit package contains some finite number of permits, and each reject package represents infinitely many rejects. A permit package may be either *static* or *mobile*. The whiteboard at a vertex w contains the set of the packages residing in w. For each such permit package P, the whiteboard contains (1) a flag indicating whether P is static or mobile, and (2) the size S of P (note, that the level of P can be calculated from its size). In addition, the whiteboard at w contains the parameter M, W and U, as well as the Boolean variable *state* which may receive either the value "locked" or the value

"unlocked". A node whose state variable is "locked" (respectively, "unlocked") is referred to as *locked* (resp., *unlocked*). The whiteboard at the root also contains the variable $Storage$, initially set to M.

The agent algorithm uses a *taxi* algorithm as a carrier. That is, in every node, the agent algorithm has an interface via which it can issue instructions to the taxi algorithm. One such instruction is to move towards the root. If the agent issues this *Up* instruction while it is in some node w that is not the root, the agent process is suspended and then awakened in the parent of w. If the agent created at u is at some ancestor w of u, a second kind of instruction it may issue to the taxi algorithm is *Down*. When the agent at w issues this *Down* instruction, its process is suspended and then awakened in the child of w on the path connecting w and u. In addition, the agent at node w may query the taxi algorithm for the distance between u and w. This is done by the command *Distance*. Another query the agent can issue is *DistToTop* which returns the distance to the topmost node (closest to the root) ever reached by this agent.

The algorithm

1. The arrival of a request at a node u (as an input from the environment) creates an agent at u to which we refer as the agent of u, though a unique identity for u is not necessary. The agent is created with one variable: Bag (initially empty). An agent that is created at an *unlocked* node, locks that node immediately, by writing "locked" in the state variable in the node's whiteboard.

   (a) If an agent is created at a node which was already locked by some other agent or reaches such a node at some later point, then the following happens.
   The agent waits passively in the locked node until the node is unlocked (by some other agent). If more than one agent is waiting for a node to become unlocked, then the agent that resumes execution (when the node becomes unlocked) is the first one waiting, according to the First In First Out discipline. (We assume that local computation take zero time, hence, once the node becomes unlocked, the first agent waiting at the node is dequeued before any new agent reaches the node from another node; this assumption causes no loss of generality since, alternatively, one can change the algorithm slightly so that a token waiting at a node gets preference over one that just arrives). When an agent is dequeued from a node's queue it continues its actions assuming it has just entered the node.

   (b) A node that contains a reject package is called a *reject* node. If the agent is created at a reject node then a reject is delivered to the request. Otherwise, if the agent of u starts moving and then reaches a reject node (for the first time), then the agent walks to its origin node u, places a reject package at every intermediate node, and delivers a reject to the request at u.

   Otherwise, if u is unlocked and does not contain a reject package, then the agent acts according to the following items.

2. If the agent is still at u (i.e, the vertex where the agent was created) and u's whiteboard contains a static package P of size S, then the following happens. A single permit in P is granted to the request as an output to the environment. Subsequently, after the request is granted, the agent rewrites the size of P in u's whiteboard to be $S-1$. If, consequently, the size of P is zero then the agent cancels P. Canceling means erasing P from u's whiteboard, together with all of P's attributes (i.e., P's size and corresponding flag). Consider the following cases.

   (a) If the granted request is for a topological event which is of type "remove leaf" or "remove internal node" and u, the vertex to be removed, holds several packages or a non-empty queue

(storing agents), then these packages and agents are first moved to u's parent (in a "graceful" manner). Subsequently, u is removed. All this is done according to the "graceful" manner discussed earlier.

(b) Otherwise (if the granted request was not to delete a node), the requested event takes place when the request is granted a permit. If the request was to add a node v (as a child of u) then u also informs v about the values M, W and U (which are needed for v in order to calculate ϕ and ψ).

After the above is performed, the agent terminates.

If there is no static package at u then the following happens.

3. The agent applies the *Up* command repeatedly until reaching either the root or a node $\rho(u)$ that is a filler node with respect to u at that time (as defined in Section 3) or some node x marked "locked", or a node that contains a reject package. To detect a filler node, whenever the agent reaches a new node w, it queries the taxi for the value $d(u, w)$ using the command *Distance*. Consider the following cases.

   (a) The case where x is marked "locked" is described in item 1a above, and the case that x contains a reject package is described in item 1b above. Otherwise (x is not marked "locked" and does not contain a reject package), the agent acts as follows.

   (b) If the node is a filler $\rho(u)$ (with respect to u) then let $P(u)$ be the mobile package of level $j(u)$ residing in $\rho(u)$ as described in the definition of a filler node above.

   (c) If the agent reaches the root and the root is not a filler node (with respect to u) then let $j(u)$ be the smallest integer $j(u) \geq 0$ such that $d(u, r) \leq 2^{j(u)+1}\psi$. Consider two cases. If the *Storage* variable is less than $2^{j(u)}\phi$ then the agent creates a reject agent, and continues only after the reject agent finished its action on the root. As described later, by that time, the root contains a reject package, and the agent acts as in item 1b. Otherwise, if the *Storage* variable is at least $2^{j(u)}\phi$, then the agent writes on the whiteboard of the root a new package $P(u)$ of size $2^{j(u)}\phi$ (the permits are taken from the root's storage) and indicates in the package's flag that $P(u)$ is mobile. Then, the agent decreases the value of the variable *Storage* (written on the root's whiteboard) by $2^{j(u)}\phi$.

4. Unless a reject agent was created in Item 3c above, the following happens.
   For each $k \in \{0, 1, 2, \cdots, j(u) - 1\}$, let u_k be the ancestor of u satisfying $d(u, u_k) = 3 \cdot 2^{k-1}\psi$. The agent applies the following procedure PROC(P) recursively, starting with the mobile package $P = P(u)$ of level $j(u)$ at its hosting node.
   PROC(P): Given a mobile package P of level k written in the whiteboard of a vertex w, act as follows.

   - Erase P from w's whiteboard and put k inside the variable *Bag* of the agent.
   - If $k > 0$ then the agent invokes the instruction *Down* repeatedly until reaching node u_k (note, that the agent can recognize u_k by querying the taxi with the instruction *Distance*, and using the value k in its *Bag* variable). Then, the agent writes in u_k's whiteboard two mobile packages P_1 and P_2, each of level $k-1$. Also, the agent empties its *Bag* variable, and applies the procedure recursively with P_2.
   - If $k = 0$ (including the case where $j(u) = 0$), then the agent invokes the instruction *Down* repeatedly, until reaching node u (note that the agent can recognize u by querying the taxi

with the instruction *Distance*).
The request is then granted to u from the level zero package P as described in item 2 above.
The agent then walks up again to the topmost node x it ever reached, using the command *Up* and the query command *DistToTop*.
When the agent reaches x, it starts moving back to u using the instruction *Down* and the query *Distance*. On its way down the tree from x to u, the agent sets the state of every node passed (including u) to "unlocked".

Reject agents: Recall, that in item 3c above, the root may create a reject agent. The reject agent at a node w with $d > 0$ children applies the following *reject procedure*: it places a reject package in the local whiteboard of w and creates d reject agents, one per child, each carrying (in a *Bag* variable) a reject package (representing infinitely many rejects). Each of these new d reject agents moves to a different child of w and applies the reject procedure recursively.

4.3.2 Implementing the taxi algorithm

The implementation is rather straightforward. Let us, however, still mention a few hints. First, when the tree is built, we assume that each node remembers its port number leading to its parent. Therefore, the *Up* command can be implemented easily by the taxi algorithm. Second, when an agent arrives at a node, we assume that the node can detect on which link the agent arrived. This information is accessible to the taxi algorithm in that node. If the agent becomes passive (waiting for the node to become unlocked) then the taxi algorithm at that node u puts the agent in the agents queue of u.

For an agent created at u, the taxi algorithm carries with the agent also the counter *Distance*, containing the value of the distance from the agent to u. In particular, when the agent is created in u, the taxi algorithm at u sets this value to zero. When the agent arrives at a new node following the execution of an *Up* action, the taxi algorithm in the new node increases this value by one. When the agent arrives at a node following a *Down*(u) operation, the taxi algorithm at the new node deducts one from this value. Counter *Distance* enables the taxi algorithm to answer the *Distance* query from the agent. When the agent is at the root or at the filler node with respect to u, then the taxi algorithm initializes its counter *DistToTop* to zero. The taxi then updates the value of *DistToTop* similarly to the way it maintains *Distance*.

When the agent locks a node, the taxi algorithm saves in the node's whiteboard the pointer to the edge leading to the child from which the locking agent arrived. (For other agents that may reside in the locked node, waiting for the node to become unlocked, their arrival ports are not remembered in the whiteboard, but are still carried in the agents.) Add to this the fact that all the nodes on the route of the agent from u to the agent's current host node are locked. Hence, the taxi algorithm has enough information to perform the *Down* command (that is, to know to which child to send the agent on the way down).

4.4 Correctness and complexity

The main idea behind proving the distributed case is to reduce it to the centralized case already proved above. Broadly speaking, we show that the actions of the distributed controller can be translated

(almost) one to one to actions of the centralized one. (This should not be too surprising, since we designed the protocol that way in order to ease its design and proof).

4.4.1 Correctness

Observation 4.1. *In any execution of the distributed controller, every agent returns to its origin eventually, and every request is answered (either by a permit or by a reject).*

The observation is rather obvious. Still, for the sake of completeness let us sketch a proof for it. Recall that agents lock nodes only on the way towards the root. Hence, no cycle can be created in the chain of agents waiting for each other. Recall also, if multiple agents wait at a locked node then, when the node becomes unlocked, the agent resuming its action, is the first one of them to have arrived at the node. Therefore, no deadlock and no starvation can arise, i.e., every request is eventually either granted a permit or rejected.

Let σ be a scenario of inputs (i.e., a sequence of requests) for the distributed controller and let $E_{Dist}(\sigma)$ be an execution of the distributed controller on scenario σ. We say that a request in σ is *relevant* with respect to the execution $E_{Dist}(\sigma)$ if the request is granted in $E_{Dist}(\sigma)$ eventually. A request in σ which is not relevant with respect to $E_{Dist}(\sigma)$ is called *irrelevant* with respect to $E_{Dist}(\sigma)$. Note that an irrelevant request with respect to $E_{Dist}(\sigma)$ is a request which is rejected eventually. Note also, that from the definition of the problem, scenario σ of inputs may consist of an infinite sequence of requests. However, the number of relevant requests in σ (with respect to $E_{Dist}(\sigma)$) is finite, as the total number of permits granted is at most M.

Lemma 4.2. *Let σ be a scenario of inputs for the distributed controller and let $E_{Dist}(\sigma)$ be an execution of the distributed controller on scenario σ. There exists a second scenario σ' and an execution $E'_{Dist}(\sigma')$ of the distributed controller on scenario σ', satisfying the following conditions.*

1. *No request is rejected in $E'_{Dist}(\sigma')$.*
2. *The final number of requests granted in the executions $E_{Dist}(\sigma)$ and $E'_{Dist}(\sigma')$ is the same.*
3. *The total number of messages used during the executions $E_{Dist}(\sigma)$ and $E'_{Dist}(\sigma')$ differs by an additive term of $O(U)$.*

Proof: The lemma follows by restricting σ to its sub-scenario σ', containing only the relevant requests in σ with respect to $E_{Dist}(\sigma)$, in their original order and time of arrivals. The execution $E'_{Dist}(\sigma')$ of σ' is the same as $E_{Dist}(\sigma)$ restricted to the actions of the controller with response to the relevant requests in σ. The fact that the first and second conditions in the lemma hold, is immediate. The fact that the third condition also holds follows from the fact that the message complexity needed to broadcast the reject packages is at most $O(U)$. ∎

Lemma 4.3. *Let σ be a scenario of inputs for the distributed controller and let $E_{Dist}(\sigma)$ be an execution of the distributed controller on scenario σ, such that no request is rejected in $E_{Dist}(\sigma)$. There exists a second scenario σ' and an execution $E'_{Dist}(\sigma')$ of the distributed controller on scenario σ', satisfying the following conditions.*

1. *Each request in σ' arrives after all the actions of the controller in $E'_{Dist}(\sigma')$ handling the previous request have been completed.*

2. *The final dynamic data structures (packages and agents) resulted by executions $E_{Dist}(\sigma)$ and $E'_{Dist}(\sigma')$ are the same.*

3. *The final number of requests granted in the executions $E_{Dist}(\sigma)$ and $E'_{Dist}(\sigma')$ are the same.*

4. *The total number of messages used during the executions $E_{Dist}(\sigma)$ and $E'_{Dist}(\sigma')$ is asymptotically the same, i.e., it differs by at most some fixed multiplicative constant.*

Proof: Let σ be a scenario of inputs for the distributed controller and let $E_{Dist}(\sigma)$ be an execution of the distributed controller on scenario σ, such that no request is rejected in $E_{Dist}(\sigma)$. Then, in particular, every request in σ is relevant.

Let $t_{max}(E_{Dist}(\sigma))$ denote the first time after which no package is moved in execution $E_{Dist}(\sigma)$. Note that all the requests in σ arrive before time $t_{max}(E_{Dist})$. Moreover, all the actions of the controller made in the execution $E_{Dist}(\sigma)$ in response to these requests are completed by time $t_{max}(E_{Dist})$.

For a request $R \in \sigma$, let $P(R)$ denote the subpath of nodes that were ever locked by the agent handling R. Note that $P(R)$ is a (non-empty) path connecting u, the *Origin* of R, with one of u's ancestors (possibly u itself). Let R_{last} be the request that is the last to receive a permit in $E_{Dist}(\sigma)$. It is immediate from the distributed implementation, that in execution $E_{Dist}(\sigma)$, request R_{last} is granted before the agent of R_{last} unlocks any node in $P(R_{last})$. Therefore, starting from the time the agent of R_{last} locked some $w \in P(R_{last})$ in execution $E_{Dist}(\sigma)$, no agent in $\mathcal{R}(E_{Dist}(\sigma))$ has been waiting in the agents queue of w. (Otherwise, that agent would have answered its corresponding request after request R_{last} was answered). Similarly, no request agent passed through w after the agent of R_{last} unlocked it.

Let s be the number of requests in σ. For $i = 1, 2, \cdots, s$, let R_i denote the i'th request (according to the order of arrival) and let t_i be the time that R_i arrived (from the environment). Scenario σ can be described as the sequence of pairs $(R_1, t_1), (R_2, t_2), \cdots, (R_s, t_s)$. Let j be the index such that R_{last} is R_j. (Note, the index j is not necessarily s.) Let $\hat{t}$ be a time after all the updates made by agents corresponding to requests $R_1, \cdots, R_{j-1}, R_{j+1}, \cdots, R_s$ were completed in execution $E_{Dist}(\sigma)$.

Consider now scenario $\hat{\sigma}$: $(R_1, t_1), (R_2, t_2), \cdots, (R_{j-1}, t_{j-1}), (R_{j+1}, t_{j+1}), \cdots, (R_s, t_s), (R_j, \hat{t})$. It follows from the above discussion that there exists an execution $\hat{E}_{Dist}(\hat{\sigma})$ of the distributed algorithm on scenario $\hat{\sigma}$ such that the following hold: (1) the last request in $\hat{\sigma}$ is initiated after all the actions of the controller in $\hat{E}_{Dist}(\hat{\sigma})$ handling the previous requests have been completed, (2) the final dynamic data structure resulted from execution $E_{Dist}(\sigma)$ is the same as the one resulted from execution $\hat{E}_{Dist}(\hat{\sigma})$, and (3) the number of requests granted and the number of messages used during these executions is the same.

The lemma follows by induction on the distance from the end of sequence $(R_1, t_1), (R_2, t_2), \cdots, (R_s, t_s)$ and repeating the above argument. ∎

The following corollary follows by combining Lemma 4.2 and Lemma 4.3.

Corollary 4.4. *Let σ be a scenario of inputs for the distributed controller and let $E_{Dist}(\sigma)$ be an execution of the distributed controller on scenario σ. There exists a second scenario σ' and an execution $E'_{Dist}(\sigma')$ of the distributed controller on scenario σ', satisfying the following conditions.*

1. *Each request in σ' is initiated after all actions of the controller in $E'_{Dist}(\sigma')$ regarding the previous request have been completed.*

2. *The final number of requests granted in the executions $E_{Dist}(\sigma)$ and $E'_{Dist}(\sigma')$ is the same.*

3. *The total number of messages used during the executions $E_{Dist}(\sigma)$ and $E'_{Dist}(\sigma')$ is the same up to an additive term of $O(U)$.*

Lemma 4.5. *Let σ be a scenario of inputs for the distributed controller and let $E_{Dist}(\sigma)$ be an execution of the distributed controller on scenario σ. Let σ' and $E'_{Dist}(\sigma')$ be a scenario and an execution such as those whose existence is claimed in Corollary 4.4. Let $E_{Cen}(\sigma')$ be the centralized execution on scenario σ'. The following two conditions hold.*

1. *The operations performed in $E'_{Dist}(\sigma')$ on the data structure are the same as the corresponding centralized operations that are performed in $E_{Cen}(\sigma')$.*
2. *Assuming each message is encoded using $O(\log N)$ bits, the message complexity used in Execution $E'_{Dist}(\sigma')$ is $O(U\frac{M}{W}\log^2 U)$.*

Proof: We prove the lemma by induction on the order of arrival of requests to $E_{Cen}(\sigma')$. The claim holds for an empty $E_{Cen}(\sigma')$ trivially. Assume now, that the claim holds for a prefix of $E_{Cen}(\sigma')$ and consider the next request R in $E_{Cen}(\sigma')$. Let u be the node to which R arrives. The lemma follows from the property of $E'_{Dist}(\sigma)'$ that no action is taken for any other request until R is answered, from Observation 4.1, from the induction hypothesis that the data structures are the same at this point for the executions $E'_{Dist}(\sigma')$ and $E_{Cen}(\sigma')$, and from following the fact that the actions taken by the agent in the distributed controller in this case are exactly the actions taken by the centralized algorithm. In particular, if the agent corresponding to R decides that its hosting node is a filler node with respect to u, then this indeed is the case, since no other agent has acted from the time this agent arrived, and therefore no topological change have occurred. Moreover, this is indeed the closest filler node to u, since otherwise the agent would have found a closer one earlier.

As for the message complexity, note that the messages are used only to move the agents. Note also, that in the case that the agent reached a filler node or the root, it traversed only four times the distance from u to the filler node. This distance is also the move complexity resulting from the arrival of the corresponding request in the centralized setting. Therefore, by Lemma 3.3, the message complexity used in Execution $E'_{Dist}(\sigma')$ is $O(U\frac{M}{W}\log^2 U)$.

The move of the data structure of a deleted node to its parent may cause many messages if the data structure is large and the message size is bounded. Let $deg(v)$ denote the *child-degree* of v, namely, the number of v's children in the tree. In Claim 4.8 below we show that the size of a data structure at a node v is always $O(deg(v)\log N + \log^3 N + \log^2 U)$. By breaking the data structure of v into $O(deg(v) + \log^2 U)$ equal size pieces of information (of size $O(\log N)$), we can move the data structure of v to its parent using $O(deg(v) + \log^2 U)$ messages of size $O(\log N)$ (note, it is not required that the value N is known in advance). Observe that the addition of a node w (either a leaf or internal) contributes 1 to the child-degree $deg(u)$ of only one node u, namely, w's parent in case w is a leaf and w itself if w is an internal node. Therefore, the total sum of the child-degrees of deleted nodes (and in fact of all nodes) is $O(U)$. Thus, the moving of the data structures of all the deleted nodes can be done using $O(U\log^2 U)$ messages of size $O(\log N)$. Note also that an agent walking up the tree at some time t, needs only to remember the number of steps k it passed from its origin. Since all the nodes on the path connecting the agent and its origin are locked, the current number n of nodes in the tree is at least k, and therefore, the agent can be encoded using $O(\log N)$ bits. A similar argument holds also for the case that the agent walks down the tree, towards its origin. It thus follows that any message can be encoded using $O(\log N)$ bits. The lemma follows. ∎

The lemma below follows directly from the above lemma and from Lemmas 3.2 and 3.3.

Lemma 4.6. *The distributed (M, W)-Controller is correct and its message complexity is $O(U\frac{M}{W}\log^2 U)$. In addition, each message is encoded using $O(\log N)$ bits.*

As in Observation 3.4, using iterations, the message complexity of the (M, W)-Controller can be reduced further. We first run the controller in iterations. In the i'th iteration, we run the terminating $(M_i, M_i/2)$-Controller, where initially, $M_1 = M$. (As shown in Observation 2.1, to get the terminating $(M_i, M_i/2)$-Controller, we run the $(M_i, M_i/2)$-Controller and if the controller wishes to reject some request then instead, we performs a simple broadcast and upcast operation (see e.g., [30] for details) for making sure that all the requested topology changes that received a permit indeed occur. We then consider the iteration is to be terminated.) When the i'th iteration terminates, $O(U)$ messages of size $O(\log M)$ suffice to count the number $L \leq M_i \leq M$ of unused permits mentioned in the proof of Observation 3.4. Recall, that $\log M = O(\log^3 n_0)$. Therefore, $O(U \log^2 U)$ messages of size $O(\log N)$ suffice to count the number L. Note that this is not more than the message complexity of the iteration. When the i'th iteration terminates, we set $M_{i+1} = L$ and run the $i + 1$'st iteration.

If $W > 0$, then, as in Observation 3.4, after $i' = O(\log \frac{M}{W+1})$ iterations, the number of unused permits in the existing packages is within a constant multiplicative factor of W. In the $i+1$'st iteration (which is also the last iteration), we run the $(M_{i'+1}, W)$-Controller, where in this iteration, we do not prevent the controller from assigning rejects.

In order to deal with the case where $W = 0$, we first run the $(M, 1)$-Controller until it terminates. If the number of permits issued by the $(M, 1)$-Controller is M then we simply place reject packages everywhere. Otherwise, if the number of permits issued by the $(M, 1)$-Controller is $M - 1$ then, we simply invoke the trivial $(1, 0)$-Controller. This sketches the proof of the following theorem which is the distributed equivalent of Observation 3.4.

Theorem 4.7. *There exists an (M, W)-Controller with message complexity $O(U \log^2 U \log \frac{M}{W+1})$, where each message is encoded using $O(\log N)$ bits.*

4.4.2 Memory space complexity

The memory space complexity of the distributed controller depends on the model to some degree. If a large number of requests can be injected by the environment at once, a node may need to use memory to hold all these requests. To account only for memory used by the algorithm, we prefer to assume that a node can refuse to accept an additional request until it finishes handling the current one.

The size of the *Bag* carried by an agent is the size of a mobile package, which is determined by its level, and therefore can be encoded using $O(\log \log N)$ bits. In addition, the taxi counts up to N for each agent, to implement *Distance* and *DistToTop*. The rest of the memory needed per agent is constant. Because of the locking mechanism, the queue at each node may contain up to one agent per child. This results in $O(deg(v) \log N)$ bits of memory at a node v (recall, $deg(v)$ is the number of v's children). Let us note however, that in the *designer port model* (see e.g., [15, 20, 25, 28]), in which the port numbers can be enumerated by the designer of the protocol, the queue of agents at a node v may be distributed among the children of v in a directed list, incurring only an extra additive term of $O(\log N)$ bits to the memory of each child of v. This saves in memory at v.

Since all mobile packages of a given level k are identical, it is enough to remember at each node v, the number of mobile level k packages it hosts, for each level k. By the domain invariants, at any given time, the number of mobile level k packages is at most U. By Lemma 4.5, this is true also in the

distributed setting. Therefore, each node u needs to remember $O(\log U)$ bits of memory for the mobile packages of a given level k. Summing over all levels k, we obtain an upper bound of $O(\log^2 U)$ bits of memory per node for letting the a node remember all the it's mobile packages.

In contrast to the mobile packages, the static packages at a node v are not identical (each contains up to ϕ permits). However, since the static packages do not move, we can actually consider all static packages at v as one combined static package. Therefore, it is enough to let v remember the total number of permits in its static packages which is at most M and therefore can be encoded using $O(\log M) = O(\log^3 N)$ bits. In addition, each node needs $O(\log M + \log U)$ bits to remember M, U and W which are required for its operation. Also, the root contains the variable *Storage*, which is of size $O(\log M)$. We therefore obtain the following claim.

Claim 4.8. *The memory required at each node v is $O(deg(v) \log N + \log^3 N + \log^2 U)$ bits.*

4.5 The case that no fixed U is known

For the distributed implementation described so far, it was assumed that a fixed bound on the number of nodes U is known in advance to all nodes. The case that a fixed U is not known is reduced again to the case that such a fixed U is given. The reduction is a rather straightforward distributed implementation of the one described in Section 3.3. For completeness, the reduction is given in Appendix A.

Theorem 4.9. *There exists a distributed (M, W)-Controller whose message complexity is $O(n_0 \log^2 n_0 \cdot \log \frac{M}{W+1} + \sum_j \log^2 n_j \cdot \log \frac{M}{W+1})$, where n_j is the number of nodes at the time the j'th topological change takes place. Moreover, each message is encoded using $O(\log N)$ bits.*

Theorem 4.9, combined with Observation 2.1, yields the following.

Corollary 4.10. *There exists a distributed terminating (M, W)-Controller whose message complexity is $O(n_0 \log^2 n_0 \cdot \log \frac{M}{W+1} + \sum_j \log^2 n_j \cdot \log \frac{M}{W+1})$, where n_j is the number of nodes when the j'th topological change takes place. Moreover, each message is encoded using $O(\log N)$ bits.*

5 Applications

In [4], the authors show how to use their (M, W)-Controller to construct several basic schemes on growing trees (allowing only leaves to join). In particular, they show how to derive a a size-estimation protocol for every fixed constant $\beta > 1$ as well as a *name-assignment* protocol. Both of these protocols incur $O(N \log^2 N)$ messages, where N is the maximum number of nodes in the tree.

In this section, we demonstrate how to use our controller to solve these problems efficiently in our extended dynamic framework. We also solve some additional related problems. Since the main techniques where already presented earlier in the paper, in this section we just describe how to apply them, and give only sketches of the proofs.

5.1 The size-estimation protocol

Consider the size-estimation problem, and let $\beta > 1$ be the required approximation parameter. We construct a size-estimation protocol that operates in iterations. Fix an iteration i and let N_i be

the number of nodes in the graph at the beginning of the iteration (for the first iteration, we have $N_1 = n_0$). The number N_i is calculated and communicated to all nodes at the beginning of the iteration. This N_i is used at each node as the estimation of the number of nodes during that iteration. (Below, we show that it indeed approximates the size well). Let $\alpha = 1 - 1/\beta$. In order to generate the next iteration, after calculating N_i, we simply run our terminating $(\alpha N_i, \alpha N_i/2)$-Controller given by Corollary 4.10. The next iteration is started when this terminating controller terminates. This guarantees that the number of nodes in the graph during the iteration is at least $N_i - \alpha N_i = (1-\alpha)N_i = N_i/\beta$ and at most $N_i(1+\alpha) = (2-1/\beta)N_i \leq \beta N_i$. Thus, during the iteration, N_i serves as a β-approximation to the number of nodes, as required. As for the message complexity, observe (by Corollary 4.10) that each iteration incurs $O(N_i \log^2 N_i)$ messages. Since an iteration doesn't terminate before $\alpha N_i/2 = \Omega(N_i)$ topological changes occurred, the total number of messages sent during the whole execution is $O(n_0 \log^2 n_0) + O(\sum_i \log^2 n_j)$. For any iteration i, and at any given time during the iteration, the number of nodes is $\Theta(N_i)$. It follows that each message is encoded using $O(\log n)$ bits, where n is the size of the graph when the message is sent. This sketches the proof of the following theorem.

Theorem 5.1. *For every constant $\beta > 1$, there exists a size-estimation protocol whose message complexity is $O(n_0 \log^2 n_0 + \sum_i \log^2 n_j)$. Moreover, each message is encoded using $O(\log n)$ bits.*

5.2 The name-assignment protocol

Let us now explain how to implement a name-assignment protocol. We assume that initially, each node has an identity in the range $[1, n_0]$. As in the case of the previous application, the protocol operates in iterations. Fix an iteration i and let N_i denote the number of nodes in the graph at the beginning of the iteration (for the first iteration, we have $N_1 = n_0$). The number N_i is calculated and communicated to all nodes at the beginning of the iteration. At this point, we would like to have for the i-th iteration, a property that is similar to the one that we had for the initial iteration. That is, we would like that the identities of all the N_i nodes are in the range $[1, N_i]$. There is some delicate point here. At first glance it seems as if this could be achieved by an agent who would have traversed the tree in a Depth First Search (DFS) manner, while assigning new DFS identities to the nodes. Unfortunately, after iteration $i-1$ (for $i > 0$) some of these identities may already be taken. Hence, performing the assignment in that DFS order, may have violated the requirement that each identity is unique at all times.

To overcome this obstacle, we first assign identities in another ("temporary") range, and only at a second stage assign the $[1, N_i]$ range. Specifically, first, an agent that is created at the root traverses the tree T in a DFS manner, and assigns each node v the identity $id(v) = 3N_i + \mathrm{DFS}(v)$, where $\mathrm{DFS}(v)$ is the DFS number associated with vertex v. When this traversal is over, a new agent traverses T and changes each identity $id(v)$ to $\mathrm{DFS}(v)$. Obviously, this guarantees that when the process is over all nodes have unique identities in the range $[1, N_i]$. As shown below, we also guarantee that at the beginning of the i'th iteration, the identities are unique and in the range $[1, 3N_i]$. Thus, the identities remain unique during the above reassignment process.

Subsequently, after the above reassignment process is completed, the root initiates the terminating $(N_i/2, N_i/4)$-Controller to take care of the naming of new nodes who join during the iteration. More precisely, the terminating controller is used, together with the following additions that calculate the new names from the variables of the controller. At all times, the permits stored at the root are represented by some interval $[A, B]$, where initially, $A = N_i + 1$ and $B = 3N_i/2$, and each integer in the interval represents a permit. Similarly, every package is encoded by some explicit interval, and the permits

in the package are associated with the integers in the interval. (The way we presented the controller above, only the number x of permits in the package was important, while now, we are using the specific x "serial numbers" of permits, organized in intervals). This is implemented as follows. When the root creates a new (mobile) package P of size S, it associates it with the interval $[A, A+S-1]$ and changes its own interval (corresponding to the root's storage) to $[A+S, B]$. This represents the fact that the permits numbered below $A+S$ no longer reside in the root. In addition, when two new packages P and P' are created as a result from a split of package P'', we split the interval I'' associated with P'' into two equal size intervals I and I', and associate I with P and I' with P'. The identity of a new node inserted into the graph is simply the integer of the corresponding permit (that is, the integer in an interval containing just one integer).

We first claim that the above protocol indeed implements a name-assignment protocol. To establish this, we first prove that when iteration i starts, every identity is an integer in the range $[1, 3N_i]$. We prove this claim by induction on i. First, observe, that the claim is trivially true for the first iteration. Assume, by induction, that the claim is true for iteration i, and let us show its validity for iteration $i+1$. After the process of assigning new identities by the two traversals of agents is completed, i.e., before the controller is initiated, each node has as identity in the range $[1, N_i]$. Since the controller assigns identities in the range $[N_i+1, 3N_i/2]$, we get that when the i'th iteration terminates, each node has an identity in the range $[1, 3N_i/2]$. Now, since the controller in iteration i assigns at most $N_i/2$ permits, we know that $N_{i+1} \geq N_i - N_i/2 = N_i/2$. Thus, at the beginning of iteration $i+1$, each node has an identity in the range $[1, 3N_{i+1}]$, as desired.

We now prove by induction that at any given time, the identities are unique, and each identity is an integer in the range $[1, 4n]$, where n is the current number of nodes in the tree. Fix an iteration i. By induction, we may assume that when the iteration starts, all identities are unique (obviously, this holds for the first iteration). We also know that when the iteration starts, each identity is an integer in the range $[1, 3N_i]$. This guarantees that during the process of assigning new identities by the two traversals of agents, each node always has a unique identity. Moreover, during this process, the number of nodes is $n = N_i$ and each identity is an integer in the range $[1, 4N_i]$. When the two DFS traversals are over (and the terminating controller is initiated), each node has a unique identity in the range $[1, N_i]$. Recall that each permit assigned by the controller is associated with a different integer in the range $[N_i+1, 3N_i/2]$. Now, during iteration i, the number of nodes n is always at least $N_i - N_i/2 = N_i/2$. This guarantees that at any time during the operation of the controller, each node has a unique identity in the range $[1, 3n]$. To sum up the above discussion, we get that at any time during the execution, each node has a unique identity in the range $[1, 4n]$.

Regarding the message complexity, recall (by Corollary 4.10) that, in each iteration i, the terminating $(N_i/2, N_i/4)$-Controller sends $O(N_i \log^2 N_i)$ messages. Since an iteration i doesn't terminate before $\Omega(N_i)$ topological changes occurred during the iteration, the total number of messages sent during the whole execution is $O(n_0 \log^2 n_0) + O(\sum_i \log^2 n_j)$. This sketches the proof of the following theorem.

Theorem 5.2. *There exists a name-assignment protocol with $O(n_0 \log^2 n_0 + \sum_i \log^2 n_j)$ message complexity.*

Remark: Observe that since a package of permits is represented by an explicit interval, and each node may store several packages, the memory size at a node may be large. Recall that in our controllers (as opposed to their usage for the name-assignment problem), the permits were implicit. This allowed us to reduce the memory at a node by combining packages together (for example, since all packages of level k are identical, we encoded s level-k packages at a given node by (s, k)). Here, this trick cannot work,

since we use the actual value of a permit, to keep all the names unique. The fact that the memory size is large has an impact on size of the messages too, but only in the case that nodes are deleted. This is because, when a node v is deleted, our controllers assume that the content of v's memory is moved to v's parent. Let us note, however, that the delivery of the memory of a deleted node to its parent is the responsibility of a different protocol (the one that handles the "graceful" deletion). Therefore, it makes sense to consider such memory-delivery messages as a part of that other protocol. We stress that if we indeed do not consider such messages at part of our name-assignment protocol, then the message size of our protocol is small, namely, $O(\log n)$ bits.

5.3 Maintaining a heavy-child decomposition

Our goal now is to show how to efficiently maintain a heavy-child decomposition of a dynamic tree. Recall, we need to maintain at each internal node v a pointer $\mu(v)$ (that may change occasionally) to one of its children in the dynamic tree T. The pointed child is called *heavy*, and every other child is called *light*. The pointers must be organized such that, at any given time, each node has $O(\log n)$ ancestors (in T) which are light.

Let us consider first, our size-estimation protocol for some fixed approximation parameter $\beta > 1$. Recall that this protocol runs in iterations. We define the *super-weight* $SW(u)$ of a node u at follows. Consider some time t during some iteration i, and consider a node u existing in the tree T at time t. The super-weight $SW(u)$ of u is the total number of descendants of u (including u itself) that existed in the tree at some point in time from the beginning of iteration i until time t (including, in particular, descendants of u that were added or deleted during that time). In particular, at the beginning of iteration i, $SW(u)$ is simply the number of descendants of u in T.

We claim that our size-estimation protocol can be modified slightly, to obtain a *subtree-estimator* protocol, which guarantees the following. At any time t during some iteration i, each node u in the existing tree holds a value $\tilde{\omega}(u)$ that is a β-approximation to the super-weight of u, i.e., we have $SW(u)/\beta \leq \tilde{\omega}(u) \leq \beta SW(u)$.

To see how such a subtree-estimator protocol can be obtained, consider an iteration i of the size-estimation protocol and let $\omega_0(v, i)$ denote the number of descendants of v in the tree at the beginning of the iteration. Just before this iteration starts, we let the root initiate a simple broadcast and upcast operation for letting each node v know the value $\omega_0(v, i)$. Now, in order to obtain the desired subtree-estimator protocol, each node monitors the packages of the size-estimation protocol which pass through it down the tree. Specifically, during the i'th iteration, each node v estimates its super-weight by $\tilde{\omega}(v, i) = \omega_0(v, i) + S$, where S is the number of permits passing down the tree via v (including v) during the iteration. Using our domain-invariants analysis, it can be shown easily that at any time during the iteration, $\tilde{\omega}(v, i)$ is indeed a β-approximation to the super-weight of v. This sketches the proof of the following lemma.

Lemma 5.3. *For any constant $\beta > 1$, there exists a subtree-estimator protocol for dynamic trees whose message complexity is $O(n_0 \log^2 n_0 + \sum_j \log^2 n_j)$. Moreover, each message is encoded using $O(\log n)$ bits.*

We now claim that our subtree-estimator protocol can be used to maintain a heavy-child decomposition. This is accomplished by maintaining a pointer at each internal node v pointing at one of v's children $\mu(v)$ such that the super-weight of each child $u \neq \mu(v)$ of v is at most 3/4 times the super-weight of v, i.e., $SW(u) \leq \frac{3}{4} SW(v)$. We show later how to maintain this property. For now,

we claim that if this property is satisfied for all pointers then the collection of pointers does induce a heavy-child decomposition. This is because, at any given time t, each node has $O(\log SW)$ ancestors (in T) which are light, where SW denotes the super-weight of the root at time t. Since we proved for the size-estimation protocol that $SW = \Theta(n)$, where n is the current number of nodes, we get that at any given time, each node has $O(\log n)$ ancestors which are light, as desired.

In order to maintain the pointers properly, we first invoke the subtree-estimator with approximation parameter $\beta = \sqrt{3}$. Then, whenever a non-root node v updates its estimate $\tilde{\omega}(v)$ for its super-weight, it inform its parent about its new estimate. (This information is not used in v's parent to update its knowledge of its own super-weight, and therefore, these extra messages may only increase the total number of messages by a factor of two.) Consider a node v. Among the estimates of v's children, v remembers the largest one only, and points its pointer towards a child $\mu(v)$ whose estimate is the largest. Let u be a child of v such that $u \neq \mu(v)$. We have $\tilde{\omega}(\mu(v)) \geq \tilde{\omega}(u)$ and thus:

$$SW(\mu(v)) \geq \frac{\tilde{\omega}(\mu(v))}{\beta} \geq \frac{\tilde{\omega}(u)}{\beta} \geq \frac{SW(u)}{\beta^2}.$$

On the other hand, we have $SW(v) > SW(\mu(v)) + SW(u)$, and thus:

$$SW(v) > (1 + \frac{1}{\beta^2})SW(u) = \frac{4}{3}SW(u).$$

In other words, this process guarantees that the super-weight of any child $u \neq \mu(v)$ of v is at most $3/4$ times the super-weight of v. This sketches the construction of the heavy-child decomposition

Theorem 5.4. *There exists a protocol maintaining a heavy-child decomposition in a dynamic tree whose message complexity is $O(n_0 \log^2 n_0 + \sum_j \log^2 n_j)$. Moreover, each message is encoded using $O(\log n)$ bits.*

5.4 Extending labeling schemes

We now show how to use our size-estimation protocol for extending various existing distributed data structures for *local queries* to dynamic settings. We consider distributed data structures in which a node u, when being asked a query, can answer it by merely inspecting its own data structure and possibly the data structure of some other node v also. Examples are routing ("which neighbor of u is the next on the route to v?"), distance (from u to v), etc. Note, however, that constructing the data structure, or maintaining it when the network graph changes, requires communication. The efficiency of such a scheme is measured in terms of the size of the memory required in each node and in terms of the number (and sizes) of messages required to update the data structure if the graph changes. Multiple such data structures are described in the literature, for static networks, or for limited dynamic networks. Using our size-estimation protocol, it is possible to extend many such data structures to operate efficiently also under controlled deletions of degree one nodes and sometimes also of other nodes.

Consider, for example, a case where deletions do not affect the correctness of the data structure. For example, deletions of degree one vertices do not affect the distance between nodes who are not deleted. Therefore, the correctness of a given static distance labeling scheme (such as those in [16]) is not affected by such deletions. At first glance, it may seem that extending such a data structure to support also deletions of that type is trivial: given such a data structure D that does not support deletions, just use it when deletions are allowed. Unfortunately, this approach is no longer efficient in terms of the size of the memory required in a node. For example, consider a large graph with optimal

label size OPT_L. If the number of nodes decreases significantly because of deletions, the optimal label size OPT_S for the resulting smaller graph may be much smaller than OPT_L. If the algorithm that maintains the labels does not take any action for deletions, then the data structure stays with labels of size OPT_L instead of the new optimal OPT_S. Using our size-estimation protocol, it is possible to recompute the data structure when the graph shrinks significantly. (Of course, recomputation can be performed after every deletion, but that would be inefficient in terms of the number of messages).

Let us now specify our extended schemes. We begin with the following two observations specifying schemes whose correctness is not affected by certain types of topological changes. For further details regarding static and dynamic labeling schemes and routing schemes, see e.g., [15, 20].

Observation 5.5. *The correctness of the following types of data structures is not affected by deletions of degree one vertices.*

- *Any exact (stretch 1) routing scheme (either labeled or name-independent) on any type of graph family which is closed under deletions of degree one nodes.*
- *Any labeling scheme on any type of graph family (closed under deletions of degree one nodes) which supports either the distance or the flow or the k-vertex connectivity functions. (E.g., the distance labeling schemes in [16] and the flow and vertex connectivity labeling schemes in [18, 21, 24]).*
- *Any nearest common ancestor (NCA) labeling scheme on trees. (E.g., the NCA labeling schemes in [8, 31].)*

Corollary 5.6. *Let π be one of the static data structures mentioned in the above observation, and let $f(n)$ be an upper bound on the size of the labels (or routing tables) used in π in a* static *n-node network. Let $\mathcal{M}(\pi, n)$ be an upper bound on the message complexity used to assign the labels (or routing tables) of π on a static n-node network. Assume, that $f(n)$ and $\mathcal{M}(\pi, n)$ are* reasonable[5] *functions. Then there exists an extended scheme supporting also controlled deletions of degree one vertices with label size $O(f(n))$ and message complexity $O(n_0 \log^2 n_0 + \mathcal{M}(\pi, n_0) + \sum_i (\log^2 n_i + \frac{\mathcal{M}(\pi, n_i)}{n_i}))$.*

Recall that the static ancestry labeling scheme on trees given by [17] uses asymptotically optimal label size. Moreover, the number of messages used to assign its labels is linear. The following corollary follows from the fact that the correctness of any ancestry labeling scheme on trees (and in particular, the scheme of [17]) is not affected by deletions of either internal nodes or leaves.

Corollary 5.7. *There exists a dynamic ancestry labeling scheme on trees supporting controlled deletions of both leaves and internal nodes, with asymptotically optimal label size and $O(n_0 \log^2 n_0 + \sum_i \log^2 n_i)$ message complexity.*

References

[1] S. Abiteboul, S. Alstrup, H. Kaplan, T. Milo, and T. Rauhe, T. Compact Labeling Scheme for Ancestor Queries. In *SIAM J. Comput.* **35** (2006), 1295–1309.

[2] Y. Afek, B. Awerbuch, E. Gafni, Y. Mansour, A. Rosén, and N. Shavit. Slide–the key to polynomial end-to-end communication. *Journal of Algorithms*, 22(1):158–186, 1997.

[5] By a reasonable function, we mean a function $f(n)$ satisfying that there exists a constant c, such that for any $n/2 < m < n$, $f(n) \leq c \cdot f(m)$. This condition is satisfied by a function of the form $f(n) = \alpha n^{\epsilon} \log^{\beta} n \log^{\gamma} \log n$, for $\alpha, \epsilon, \beta, \gamma > 0$.

[3] Y. Afek, B. Awerbuch, and E. Gafni. Applying static network protocols to dynamic networks. In *Proc. 28th Symp. on Foundations of Computer Science (FOCS)*, pages 358–370, 1987.

[4] Y. Afek, B. Awerbuch, S.A. Plotkin, and M. Saks. Local management of a global resource in a communication network. *J. ACM* **43**, (1996), 1–19.

[5] Y. Afek, E. Gafni, and M. Ricklin. Upper and lower bounds for routing schemes in dynamic networks. In *Proc. 30th Symp. on Foundations of Computer Science (FOCS)*, 1989, 370–375.

[6] Y. Afek, and M. Ricklin. Sparser: a paradigm for running distributed algorithms *J. of Algorithms* **14**, (1993), 316–328.

[7] Y. Afek and M. E. Saks. Detecting Global Termination Conditions in the Face of Uncertainty. In *Proc. 6th Ann. ACM SIGACT-SIGOPS Symp. on Principles of Distributed Computing (PODC)* ,1987, pp. 109-124.

[8] S. Alstrup, C. Gavoille, H. Kaplan and T. Rauhe. Nearest Common Ancestors: A Survey and a new Distributed Algorithm. *Theory of Computing Systems* **37**, (2004), 441–456.

[9] R. Bar-Yehuda and S. Kutten. Fault-Tolerant Majority Commitment. J. of Alg. Vol. 9, pp. 568–582, 1988.

[10] L. Barrire, P. Flocchini, P. Fraigniaud, and N. Santoro. Can we elect if we cannot compare? In *Proc. 15th ACM Symp. on Parallelism in Algorithms and Architectures (SPAA)*, 2003, pp. 324-332.

[11] Y. Bartal and A. Rosen. The Distributed k-Server Problem- A Competitive Distributed Translator for k-Server Algorithms. In *Proc. 33rd Symp. on Foundations of Computer Science (FOCS)*, pp. 344-354, 1992.

[12] E. W. Dijkstra. A note on two problems in connexion with graphs. Numerische Math. 1 (1959), S. pp. 269-271.

[13] M. Elkin. A near-optimal distributed fully dynamic algorithm for maintaining sparse spanners. In *Proc. 26th ACM Symp. on Principles of Distributed Computing (PODC)*, pages 185–194, 2007.

[14] M. J. Fischer, N. A. Lynch, and M. Paterson. Impossibility of Distributed Consensus with One Faulty Process. J. ACM 32(2), pp. 374-382 (1985).

[15] P. Fraigniaud and C. Gavoille. Routing in trees. In *Proc. 28th Int. Colloq. on Automata, Languages, and Programming (ICALP)*. 2001, pp. 757–772.

[16] C. Gavoille, D. Peleg, S. Pérennes, and R. Raz. Distance labeling in graphs. *J. of Algorithms* **53(1)** (2004), 85–112.

[17] S. Kannan, M. Naor, and S. Rudich. Implicit representation of graphs. In *SIAM J. on Descrete Math* **5**, (1992), 596–603.

[18] M. Katz, N.A. Katz, A. Korman, and D. Peleg. Labeling schemes for flow and connectivity. *SIAM Journal on Computing* **34** (2004), 23–40.

[19] E. Korach, S. Kutten, and S. Moran. A Modular Technique for the Design of Efficient Distributed Leader Finding Algorithms. ACM Transactions on Programming Languages and Systems, Vol. 12, No. 1, pp. 84–101, 1990.

[20] A. Korman. General Compact Labeling Schemes for Dynamic Trees. J. Distributed Computing **20(3)**, 179-193 (2007).

[21] A. Korman. Labeling Schemes for Vertex Connectivity. ACM Transactions on Algorithms **6(2)** (2010).

[22] A. Korman. Improved Compact Routing Schemes for Dynamic Trees In *Proc. 27th Ann. ACM SIGACT-SIGOPS Symp. on Principles of Distributed Computing (PODC)*, 2008.

[23] A. Korman. Compact Routing Schemes for Dynamic Trees in the Fixed Port Model. In *Proc. 10th Int. Conf.on Distributed Computing and Networking, (ICDCN)*, 2009.

[24] A. Korman and S. Kutten. Distributed Verification of Minimum Spanning Trees. J. Distributed Computing 20(4): 253-266 (2007).

[25] A. Korman, D. Peleg, and Y. Rodeh. Labeling schemes for dynamic tree networks. Theory of Computing Systems (ToCS) 37 (2004), pp. 49-75.

[26] A. Korman and D. Peleg. Labeling Schemes for Weighted Dynamic Trees. J. Information and Computation 205(12): 1721-1740 (2007).

[27] A. Korman and D. Peleg. Dynamic Routing Schemes for Graphs with Low Local Density. ACM Transactions on Algorithms 4(4) (2008).

[28] A. Korman and D. Peleg. Compact Separator Decomposition for Dynamic Trees and Applications. J. Distributed Computing 21(2): 141-161 (2008).

[29] Z. Lotker, B. Patt-Shamir, and A. Rosén. Distributed approximate matching. In *Proc. 26th ACM Symp. on Principles of Distributed Computing (PODC)*, pages 167–174, 2007.

[30] D. Peleg . *Distributed computing: A Locality-Sensitive Approach. SIAM*, 2000.

[31] D. Peleg. Informative labeling schemes for graphs. In *Proc. 25th Symp. on Mathematical Foundations of Computer Science*, volume LNCS-1893, pages 579–588. SV, Aug. 2000.

[32] S. Kutten. Optimal Fault-Tolerant Distributed Construction of a Spanning Forest. Information Processing Letters, Vol. 27, pp. 299–307,May 1988.

[33] M.S. Manasse, L.A. McGeoch, and D.D. Sleator. Competitive Algorithms for On-Line Problems. In *Proc. 20th Annual ACM Symposium on Theory of Computing (STOC)*, 1988, pp. 322-333.

[34] J. Moy - 1994 - RFC 2328, April 1998.

[35] D. Peleg and E. Upfal. A tradeoff between size and efficiency for routing tables. *J. ACM* **36**, (1989), 510–530.

[36] R. E. Tarjan. Data Structures and Network Algorithms. *SIAM*, Philadelphia, PA, USA, 1983.

Appendix

A The case that no fixed U is known in the distributed implementation

We now describe how to handle the general case, where it is not assumed that an upper bound U on the number of nodes ever existing in the graph is given in advance. We present a distributed implementation of the algorithm described in Subsection 3.3 and the proof of the first part of Theorem 3.5. The idea is similar to the one described in Section 5 of [4].

Recall, that the centralized algorithm in the proof of the first part of Theorem 3.5 is executed in iterations. The distributed implementation of one iteration is just the distributed implementation described above.

Similarly to the centralized implementation of the general case, in the i'th iteration, we assume that $U = U_i$, and execute the terminating (M_i, W)-Controller given by Corollary 4.10, where U_i and M_i are defined below. The first iteration is initiated with $M_1 = M$, and with $U_1 = 2n_0$, where n_0 is the initial number of nodes in the graph. Let N_i denote the number of nodes in the graph at the beginning of the i'th iteration and let Y_i denote the number of (granted) requests during the i'th iteration. Let $U_i = 2N_i$ and let Z_i be the number of *topological changes* that occur during the i'th iteration.

Recall (from the centralized implementation of the general case), that an iteration is stopped when Z_i- the number of topological changes, reaches $U_i/4$. It costs many messages to detect that event distributively. Fortunately, it is easy to show that the theorem also holds if an iteration is stopped when Z_i satisfies $U_i/4 \leq Z_i \leq U_i/2$. To detect this, we employ, in parallel to the invocation of the terminating (M_i, W)-Controller, a terminating $(U_i/2, U_i/4)$-Controller that counts only the topological changes. We note that in each iteration, in order to implement the two controllers simultaneously, the agents of each controller ignore the locks of the other controller. However, a topological change occurs only after the corresponding request receives a permit from both controllers.

In particular, just before terminating, a terminating controller performs a broadcast and upcast operation (initiated by the root). This verifies that all the permitted events indeed took place before terminating. Consider an event at some node u that received a permit from one controller, but not from the other. Until receiving a permit from the other controller, the event cannot take place. This may delay the upcast (of the broadcast and upcast) of the other controller.

The above mentioned broadcast and upcast is used for two goals. One is to ensure that the termination at the root is made indeed after all the permitted events already took place. The other purpose is to prepare for the next iteration. That is, when the terminating $(U_i/2, U_i/4)$-Controller terminates, a broadcast and upcast operation is invoked by the root for calculating the current number of nodes N_{i+1} in the graph, and Y_i- the exact number of permits granted. In addition, the algorithm resets the data structure at all the nodes using another broadcast and upcast operation. The algorithm is then ready for the new iteration, in which a new terminating (M_{i+1}, W)-Controller and a new terminating $(U_{i+1}/2, U_{i+1}/4)$-Controller are invoked with parameters $U_{i+1} = 3N_{i+1}/2$ and $M_{i+1} = M_i - Y_i$.

Note, that the message complexity of the broadcast and upcast operations is at most a constant factor times the message complexity used in the iteration. In addition, the fact that we are running also the terminating $(U_i/2, U_i/4)$-Controllers in each iteration, increases the message complexity only by a constant factor, at most. Theorem 4.7 states that the message complexity used in an iteration is $O(U \log^2 U \log \frac{M}{W+1})$, assuming that each message is encoded using $O(\log U)$ bits. Since the value U

used in a given iteration satisfies $U = O(N_i) = O(N)$, where N_i is the initial number of nodes in the iteration, we obtain that each message is encoded using $O(\log N)$ bits. Moreover, we obtain that the number of messages (of size $O(\log N)$) used in iteration i is $O(N_i \log^2 N_i \log \frac{M}{W+1})$.

The above discussion sketches the proof for Theorem 4.9 that is the distributed equivalent of the first part of Theorem 3.5.